TAMPEREEN TEKNILLINEN YLIOPISTO

TEPPO LAINIO

KOMPONENTTIKEHYKSEN KEHITYS JA KÄYTTÖÖNOTTO DYNAMICS CRM KEHITYSTYÖN TUEKSI

Diplomityö

Tarkastaja: Professori Kari Systä
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunnan tiedekuntaneuvoston
kokouksessa 6.4.2016

TIIVISTELMÄ

TEPPO LAINIO: Komponenttikehyksen kehitys ja käyttöönotto Dynamics CRM kehitystyön tueksi
Tampereen teknillinen yliopisto
Diplomityö, 81 sivua, 13 liitesivua
Joulukuu 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Professori Kari Systä

Avainsanat: komponenttikehys, Dynamics CRM, ohjelmistokehys, läpileikkaava huolenaihe, konfiguraation hallinta

Yhtenäisillä suunnitteluratkaisuilla pystytään vaikuttamaan ohjelmistokehityksen tuloksena syntyvän järjestelmän virheettömyyteen ja ylläpidettävyyteen. Tässä diplomityössä haetaan ratkaisuja helpottamaan Microsoft Dynamics CRM -alustan päälle rakennettavien järjestelmien kehitystyötä. Tavoitteena on nopeuttaa kehitystyötä ja vähentää puutteellisten kehityskäytäntöjen aiheuttamia ylläpidettävyysongelmia.

Näihin tarpeisiin vastattiin rakentamalla komponenttikehys Dynamics CRM -kehitystyöhön. Komponenttikehys tukee järjestelmään asennettavien .NET-toteutusten, kuten liitännäisten, kehitystyötä. Toteutukset asennetaan järjestelmän sisään kirjastotiedostoina. Diplomityössä kartoitettiin komponenttikehykselle asetettavat vaatimukset ja tehtiin nämä vaatimukset täyttävä toteutus. Komponenttikehyksen toteutus muodostuu asetetut vaatimukset täyttävästä arkkitehtuurista ja ohjelmakoodista sekä valmiin komponenttikehyksen konfiguraation hallinnasta. Komponenttikehyksen käyttäjinä ovat yrityksen tekniset arkkitehdit ja ohjelmistokehittäjät.

Komponenttikehyksen avulla on onnistuttu vastaamaan sekä diplomityön että yrityksen liiketoiminnan tavoitteisiin. Komponenttikehyksen arkkitehtuuri ohjaa sen käyttäjiä ylläpidollisesti parempien ratkaisujen käyttöön. Komponenttikehyksen käyttöönotto on erittäin nopeaa verrattuna aiempiin käytäntöihin. Hyötyjen näkymiseen on mennyt huomattavasti ennakoitua pitempi aika. Tähän on vaikuttanut käyttöönoton muutosvastarinta. Suosittelen komponenttikehyksen kaltaisen ratkaisun käyttämistä Dynamics CRM -kehitystyössä. Lisäksi suosittelen käyttöönottoprosessin hiomista mahdollisimman sulavaksi muutosvastarinnan vähentämiseksi.

ABSTRACT

TEPPO LAINIO: Development and deployment of framework for supporting Dynamics CRM development
Tampere University of Technology
Diplomityö, 81 pages, 13 Appendix pages
December 2018
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Kari Systä

Keywords: framelet, Dynamics CRM, framework, cross-cutting concern, configuration management

In software development correctness and maintainability qualities of the resulting system can be affected with handling of the cross-cutting concerns. This master's thesis searches of ways to improve development speed when systems are built on top of Dynamics CRM platform. Aim is also to search ways to reduce supportability and maintainability issues on built systems caused by fluctuating cross-cutting concern handling on system development.

A framelet was built to support these needs. Built framelet helps on development of .NET libraries installed inside Dynamics CRM as a plugins or custom workflow activities. Necessary requirements were first gathered. An implementation then was done to fulfill these requirements. Implementation consist of framelet architecture, code implementing the architecture, and configuration management of the framelet. Users of framelet are company's software developers and technical architects.

This master's thesis has met targets set to it. Framelet has improved performance on targets set by company. Framelet architecture steers its users towards solutions offering improved maintainability. Built framelet offers a very swift way to start development work when compared to previously used habits. Performance improvement on the targets set by the business has taken much more time than was anticipated. Change resistance against usage of the framelet has affected this. I recommend usage of solutions similar to the framelet on Dynamics CRM development work. I also suggest honing the deployment process of such solution to be as smooth as possible to reduce amount of change resistance.

ALKUSANAT

Tämä työ on tehty Accountor Enterprise Solutions Oy:ssä. Accountor Enterprise toimittaa yrityksille yritysten tietojärjestelmiä liiketoiminta-alustojen avulla. Diplomityön alkaessa yrityksen nimi oli Mepco Oy.

Diplomityön idea syntyi työskennellessäni ohjelmistokehittäjänä tietyssä asiakasprojektissa. Tämän projektin yhteydessä havahduin siihen, miten järjestelmän läpileikkaavat asiat on mahdollista esittää ohjelmakoodissa niin, että esitysmuoto tukee järjestelmän operointia ja ylläpitoa. Tämä hetki toimi itselleni lähtökohtana matkalle, jolla olen oppinut paljon ohjelmistotuotannon teorioiden, ihmisten persoonien ja yritystoiminnan tavoitteiden yhteensovittamisesta. Ilokseni huomasin tämän matkan aikana, miten hyvin Tampereen teknillisen yliopiston tarjoamat opinnot antoivat teoreettista ponnistusalustaa todellisten ongelmien ratkaisemiseksi.

Haluan kiittää ohjaajanani toiminutta Kari Systää työni selkeyden kannalta erittäin arvokkaista huomioista. Haluan kiittää Karia myös erittäin hienosta asenteesta, jossa kellonajalla ei ollut juurikaan väliä vastatessasi kysymyksiini.

Kiitos Aki Hämäläiselle alkuperäisen idean esittämisestä, Ilkka Martikkalalle diplomityöidean välittömästä kannattamisesta ja Joni Talvitielle tuesta liiketoimintanäkökulman syventämisessä. Kiitos myös muille Accountor Enterprisen entisille ja nykyisille työntekijöille, joiden kanssa olen ideoinut diplomityöni toteutusta ja jotka olette haastaneet minua näistä ideoista.

Lisäksi haluan kiittää vanhempiani antamastanne tuesta opintojeni aikana. Olisin toivonut teistä molempien näkevän tämän diplomityön. Kiitos Johannalle työn oikeinkirjoituksen tarkistamisesta ja pitkittyneen diplomityöprosessini tukemisesta.

Tampereella 21.11.2018

Teppo Lainio

SISÄLLYS

KUVALUETTELO

OHJELMALUETTELO

TERMIT JA NIIDEN MÄÄRITELMÄT

CRM	Asiakkuudenhallintajärjestelmästä yleisesti käytetty lyhennys. Tulee englannin kielen sanoista Customer Relationship Management.
DAO	Olio, joka abstrahoi ja kapseloi tarvittavan logiikan jossain tietyssä tietolähteessä olevan tiedon noutamiseksi ja tallentamiseksi. Tulee englannin kielen sanoista *Data Access Object*. [97; 35, s. 596]
Dekompositio	Osiin pilkkominen.
DTO	Olio, joka kapseloi kahden järjestelmän välillä kuljetettavan tiedon. Tulee englannin kielen sanoista *Data Transfer Object*. [98; 11]
GAC	.NET-kehyksen käyttämä laitekohtainen ja koko laitteen laajuinen sijainti, josta voidaan ladata kirjastoja dynaamisesti. Termi on lyhennys englannin kielen sanoista *Global Assembly Cache*. [55; 81]
Hiekkalaatikko	Turvallisuusmekanismi, jonka avulla saadaan estettyä sovelluksen pääsy niihin osiin järjestelmästä, joihin sille ei ole annettu oikeuksia. Tulee englannin kielen termistä *sandbox*. [18]
IntelliSense	Visual Studio -kehitysympäristössä oleva koodin täydennystyökalu, joka ehdottaa sopivia muuttujia ja metodeja ohjelmistokehittäjälle [95]. IntelliSense-dokumentaatiota voi kirjoittaa myös omiin toteutuksiin [92].
Kupliminen	Ohjelman suorituksen peruuttaminen ylöspäin sen kutsuhierarkiassa sen jälkeen kun ohjelmassa on heitetty poikkeus [106, s. 303].
Käyttäjä	Komponenttikehystä asiakasprojektin toteutuksessa käyttävä Accountor Enterprise Solutions Oy:n tekninen arkkitehti tai ohjelmistokehittäjä.
Liitännäinen	Sovelluksen laajennuspisteeseen ajonaikaisesti liitettävä laajennus, joka muuttaa tai laajentaa sovelluksen toimintaa ajonaikaisesti. Termin englanninkielinen vastine on *plug-in*. [36, s. 89]
LINQ	.NET-kehyksen yleiskäyttöinen kyselykielirakenne tiedon kyselemiseksi monista eri tietolähteistä. Termi on lyhennys englanninkielisestä nimestä *.NET Language-Integrated Query*. [7]
Loppukäyttäjä	Asiakasprojektin lopputuloksena syntynyttä Dynamics CRM -järjestelmää käyttävä asiakasyrityksen työntekijä.
NuGet	NuGet on .NET-ohjelmointikielien käyttämä mekanismi, jolla ohjelmakoodiin voidaan lisätä viittaus ulkoisessa tietovarastossa olevaan binäärimuotoiseen kirjastoon. Tarvittavat kirjastot ladataan automaattisesti NuGet-ohjelmiston toimesta. [38]

Profilointi	Profilointi on ohjelmakoodin suorituksen ajonaikaista monitorointia, jossa jäljitetään tiettyjen suorituspolkujen suoritusmäärää ja suoritusten kestoa. Profilointi on tyypillisesti ohjelmakoodin optimointia tukeva työkalu. [6]
SaaS	Pilvipalvelu, jossa ohjelmiston käyttö ostetaan palveluna palvelun tarjoajan huolehtiessa kaikesta ohjelmistoon ja infrastruktuuriin liittyvästä ylläpidosta. Termi on lyhennys englannin kielen sanoista *Software as a Service*.
SDK	Kokoelma ohjelmistokehitystyökaluja sovellusten toteuttamiseksi jollakin tietyllä ohjelmistolla, ohjelmistokehyksellä, käyttöjärjestelmällä tai muulla vastaavalla kehitysalustalla. Tulee englannin kielen termistä *Software Development Kit*. [109, s. 934]
Sovitin	Suunnittelumalli, jonka avulla komponentti voi kutsua toisen komponentin yhteensopimatonta rajapintaa ilman sen tuntemista. Yhteensopimattomat rajapinnat muutetaan sovittimella (eng. *adapter*) yhteensopiviksi. [14, s. 139; 27, s. 90]
Takaisinkutsu	Mekanismi, jolla palvelun kutsuja voi saada kontrollin palvelun aikana. Takaisinkutsu mahdollistaa yleiskäyttöiset komponentit, joita voidaan muokata käyttäjän tarpeeseen. [27, s. 85]
Tehdas	Suunnittelumalli luontiriippuvuuden purkamiseen. Sovellus saa tehtaan avulla luotua uusia tietyn tyyppisiä instansseja ilman tietoa siitä, miten nämä instanssit luodaan. [27, s. 92]

1. JOHDANTO

Ohjelmistokehityksessä törmätään usein koko ohjelmiston läpileikkaaviin haasteisiin. Näiden yhtenevä ja keskitetty hallinta tai sen puute on asia, jolla pystytään vaikuttamaan ohjelmistokehityksen tuloksena syntyvän järjestelmän tekniseen virheettömyyteen. Loppukäyttäjille tämä näyttäytyy heidän tarpeisiinsa paremmin sopivana järjestelmänä, koska kehitystyötä on ohjattu läpileikkaavien huolenaiheiden hallinnan asemasta heitä auttaviin asioiden suunnitteluun ja toteutukseen.

Järjestelmän läpileikkaavien haasteiden huomiointi tuottaa herkästi monimutkaisen ja sekavan toteutuksen [25]. Tällaisten haasteiden ratkaisu tulee pyrkiä kuvaamaan yhtenä moduulina, jotta ratkaisun uudelleenkäytettävyys ei kärsi [4]. Ohjelmistokehyksillä pyritään tarjoamaan uudelleenkäytettäviä ratkaisuja joltakin tietyltä osaamisalueelta [12].

Tässä diplomityössä on tavoitteena helpottaa Microsoft Dynamics CRM -asiakkuudenhallintajärjestelmän päälle tehtävää kehitystyötä. Tämä diplomityö tehtiin Accountor Enterprise Solutions Oy:lle. Yrityksessä haettiin lyhyellä aikavälillä kehitystyön nopeuttamista ja hieman pidemmällä aikavälillä kehityskäytäntöjen yhtenäistymistä. Yrityksen sisäisenä kohderyhmänä olivat ohjelmistokehittäjät ja tekniset arkkitehdit. Lähtötilanteessa yhteisiä kehityskäytäntöjä ei ollut. Kehityskäytäntöjen puute näkyi joskus myös asiakkaille. Erityisesti tämä puute näkyi hyvin vaihtelevan tasoisena virhetilanteiden hallintana sekä yleisinä ylläpidollisina ongelmina. Yrityksen tavoitteina oli saada parannusta näihin puutteisiin parantamalla kehityskäytäntöjä.

Dynamics CRM on Microsoftin Dynamics-tuoteperheeseen kuuluva asiakkuudenhallintajärjestelmä. Microsoft tarjoaa tuotetta sekä pilvipohjaisena SaaS-palveluna että asiakkaan omille palvelimille asennettavana versiona. Tuotteen käyttöliittymä on selainpohjainen. Tuotteen muut osat ovat tietokanta ja web-palvelin. Tässä diplomityössä keskityttiin helpottamaan web-palvelimen sisään asennettavien Dynamics CRM -komponenttien kehitystyötä.

Diplomityössä rakennettiin komponenttikehys yrityksen sisäiseen käyttöön. Komponenttikehys on ohjelmistokehys, jonka erikoistaminen ei tuota itsenäistä sovellusta vaan yksittäisen komponentin [27]. Rakennettavalla komponenttikehyksellä haluttiin mahdollistaa hyväksi havaittujen tapojen yhtenäinen käyttö kaikissa yrityksen toimittamissa Dynamics CRM -järjestelmissä. Hyväksi havaitut tavat liittyvät niin järjestelmän läpileikkaavien asioiden käsittelyyn kuin myös ohjelmistokehittäjien päivittäisen työn nopeuttamiseen valmiiden toteutusten avulla.

Komponenttikehyksen arkkitehtuurin suunnittelun ja toteutuksen lisäksi diplomityössä piti huomioida konfiguraation hallinta. Komponenttikehyksen haluttiin toimivan yhtenevästi riippumatta siitä, onko käytössä Dynamics CRM:n pilvipohjainen vai paikallisille palvelimille asennettu versio. Lisäksi toiminnoista haluttiin yhtenevät sekä uusille että vanhoille Dynamics CRM -versioille. Diplomityön julkaisuhetkellä komponenttikehys tukee neljää eri Dynamics CRM -versiota. Näitä kaikkia tuetaan sekä pilvessä että paikallisesti asennettuna. Kaikki nämä versiot haluttiin tarjota yrityksen sisällä nopeasti, helposti ja automatisoidusti käyttöönotettavaksi. Käyttöönottajina ovat tekniset arkkitehdit ja ohjelmistokehittäjät.

Luvussa 2 taustoitetaaan tarkemmin se yritys- ja järjestelmäympäristö, johon diplomityö tehdään. Lisäksi kuvataan yrityksen liiketoiminnasta tulevat tavoitteet. Luvussa 3 kuvataan komponenttikehyksen arkkitehtuurin taustalla oleva teoria sekä Dynamics CRM -alustan arkkitehtuuria tämän työn kannalta oleellisin osin. Luvussa 4 kuvataan komponenttikehyksen haluttu toiminnallisuus sekä suunnittelulle asetetut reunaehdot. Luvussa 5 käydään läpi komponenttikehyksen toteutunut rakenne, julkiset rajapinnat ja käyttäminen asiakasprojekteissa. Luvussa 6 käydään läpi, miten komponenttikehyksen konfiguraation hallinta on toteutettu ja miten haluttu konfiguraatio on ohjelmistokehittäjien saatavilla. Komponenttikehyksen kehitysprosessi käydään läpi luvussa 7. Kehitysprosessi muodostuu toteutuksesta ja sen varmentamisesta sekä käyttöönotosta. Luvussa 8 tarkastellaan tuloksena syntynyttä komponenttikehystä sekä teorian että vaatimusten näkökulmasta. Lopuksi luvussa 9 vedetään yhteen tämän diplomityön johtopäätökset.

2. LÄHTÖKOHDAT

Tässä luvussa esitellään yritys, johon työ on tehty, sekä kuvataan diplomityön tekemisen lähtökohdat ja tavoitteet yrityksen näkökulmasta. Lisäksi luvussa esitellään sovellusympäristö, jonka yhteyteen diplomityön tuloksena syntyvä komponenttikehys on toteutettu.

2.1 Ympäristö

Diplomityö on toteutettu helpottamaan Microsoft Dynamics -tuoteperheeseen kuuluvaan Microsoft Dynamics CRM asiakkuudenhallintajärjestelmään tehtävää kehitystyötä. Työ on tehty suomalaisen Accountor Enterprise Solutions Oy nimisen ohjelmistoyrityksen käyttöön. Kun tätä diplomityötä aloitettiin tekemään, kyseinen yritys oli nimeltään Mepco Oy. Accountor Enterprise Solutions Oy toimittaa Dynamics CRM -järjestelmiä asiakkaiden liiketoiminnan prosessien tukemiseen. Asiakkuudenhallintajärjestelmistä käytetään yleisesti lyhennettä CRM eli Customer Relationship Management, josta myös järjestelmän nimi juontaa juurensa. Microsoft tarjoaa Dynamics CRM -tuotetta sekä pilvipohjaisena SaaS-ratkaisuna että asiakkaan omaan ympäristöön paikallisille palvelimille asennettavana versiona. Diplomityössä on tarkoitus tukea sekä paikallisia Dynamics CRM asennuksia että pilvipohjaisia Dynamic CRM instansseja. Työn tuloksena syntyvän komponenttikehyksen on tarkoitus toimia Dynamics CRM 2013 ja sitä uudemmissa Dynamics CRM -ympäristöissä. Komponenttikehyksen käyttäjinä tulevat olemaan Accountor Enterprise Solutions Oy:n työntekijöinä työskentelevät ohjelmistokehittäjät ja tekniset arkkitehdit.

Dynamics CRM -tuotteen käyttöliittymä on selainpohjainen ja sitä ajetaan Microsoftin oman www-palvelimen päällä. Tuotteen alustan taustalla on Microsoft SQL Server -tietokantapalvelimella sijaitseva tietokanta. Alusta ei tue suoraa tietokantataulujen käyttöä, vaan tarjoaa tiedon hakuun, muokkaamiseen ja muihin operaatioihin web service -rajapinnan. Alustan tietokantataulut on abstrahoitu entiteeteiksi ja niistä käytetään johdonmukaisesti tätä nimeä. Alustan käyttämän entiteetin määritelmän [60] voi tämän diplomityön osalta mieltää toteutustasolla tietokannan tauluksi.

Dynamics CRM -tuote tukee alustana laajasti erilaisia konfiguroimalla tehtäviä muutoksia järjestelmän käytön ja elinkaaren aikana. Tällaisia muutoksia ovat esimerkiksi entiteettien, kenttien ja entiteettien välisten suhteiden lisäykset, muutokset sekä poistot. Alusta mahdollistaa myös käyttöliittymän muokkaukset sekä erilaiset yksinkertaiset konfiguroimalla tehtävät automaatiot.

Sellaisiin automaatiotarpeisiin, joita ei pysty tekemään alustan tarjoamilla vakiotyökaluilla, on tarjolla tarpeesta riippuen useita erilaisia vaihtoehtoja. Palvelinpäässä ajettaviin laajennustarpeisin voidaan hyödyntää Dynamics CRM -alustan käyttämää tapahtumankäsittelylinjastoa [61]. Tämän linjaston tapahtumiin on mahdollista kiinnittää automaation suorittavia, tietyn rajapinnan toteuttavia luokkia alustan erilaisiin tapahtumiin [61]. Tältä osin laajennusarkkitehtuuri voidaan kuvata liitännäisarkkitehtuuriksi [36, s. 89]. Dynamics CRM:n terminologiassa näistä luokista käytetään arkkitehtuurin mukaista nimeä liitännäinen [71]. Toinen Dynamics CRM -alustan tarjoama vaihtoehto räätälöityihin automaatioitarpeisiin on tehdä ohjelmoimalla räätälöityjä työnkulun aktiviteetteja [61; 48]. Räätälöityjä työnkulun aktiviteetteja ajetaan osana Dynamics CRM -alustan tarjoamia eri tyyppisiä prosesseja [48; 46]. Prosessit voivat olla työnkulkuja, dialogeja tai action-tyyppisiä [48; 46]. Action-tyyppinen prosessi mahdollistaa vakion Dynamics CRM:n web service -rajapinnan laajentamisen uusilla verbeillä [46]. Jokainen näistä kolmesta on räätälöidyn työnkulun toteutusteknisestä näkökulmasta identtinen.

Totetusteknisesti liitännäisten ja räätälöityjen työnkulun vaiheiden toteuttaminen tehdään lähes identtisesti. Kummallakin tavalla tehtäessä ohjelmistokehittäjä toteuttaa *Execute()*-metodin. Tämä metodi saa parametrina Dynamics CRM:n tapahtumankäsittelylinjaston suorituskontekstin. Näiden kahden laajennustekniikan eroja on esitelty tarkemmin kohdassa 3.4.

Järjestelmän arkkitehtuurin näkökulmasta näiden kahden laajennustekniikan välillä on eroja, jotka eivät kuitenkaan ole merkityksellisiä tämän diplomityön yhteydessä. Tyypillisesti näillä kummallakin laajennustekniikalla tehdään järjestelmän kokonaisuuden kannalta yksittäisinä toteutuksina melko pieniä laajennoksia. Yksittäisen laajennoksen toteutustyö on tyypillisesti korkeintaan muutamia henkilötyöpäiviä. Isommat toiminnalliset kokonaisuudet muodostetaan tyypillisesti toteuttamalla useampia yksittäisiä pienehköjä laajennoksia, jotka yhdessä saavat aikaan halutun lopputuloksen.

2.2 Tausta ja tavoitteet

Komponenttikehyksen kehitystyöllä oli kaksi päätavoitetta. Yksi tavoite oli nopeuttaa liitännäisten ja räätälöityjen työnkulun aktiviteettien kehitystyötä. Hieman pidemmän aikavälin tavoite oli Accountor Enterprise Solutions Oy:n Dynamics CRM -kehityskäytäntöjen yhtenäistäminen. Komponenttikehyksen syntyidean aikoihin valtaosa yrityksen Dynamics CRM -projekteista oli ollut kohtuullisen pieniä ja yksittäisessä projektissa ei useinkaan tarvittu suuria määriä ohjelmoituja automaatioita. Ohjelmoitujen automaatioiden tekoon ei myöskään ollut yhtenäisiä kehitysohjeita, ellei oteta huomioon Microsoftin Dynamics CRM -kehitysopasta [74], jossa mainittuja reunaehtoja [75] tulee noudattaa. Koska kehitysohjeita ei ollut, jokainen ohjelmistokehittäjä noudatti käytännössä omaa yksilöllistä kehityskäytäntöään. Nämä yksilölliset käytännöt sisälsivät yleensä vähintään erilaisia apukirjastoja lokitukseen. Usein apukirjastoja oli tehty myös tiedon hakemiseen Dynamics CRM:stä. Lisäksi käytännössä kaikilla ohjelmistokehittäjillä oli tapana kopioida käyttämänsä koodipohjat jostain käsillä olleesta vanhasta asiakasprojektista.

Kaikki nämä käytännöt olivat kehitystyön haaste. Pienissä yhden ohjelmistokehittäjän projekteissa ongelmat eivät nousseet suuriksi, sillä asiakkailla ei usein ollut suuria vaatimuksia tai odotuksia esimerkiksi virhetilanteiden käsittelyn suhteen. Pienissä projekteissa pahimmat ongelmien esiintymiskohdat tulivat usein koodin omistuksen kautta: kukaan toinen ohjelmistokehittäjä ei tuntenut toisen kehityskäytäntöjä ja tämän takia muutoksiin haluttiin usein tekijäksi alkuperäinen ohjelmistokehittäjä. Suuremmissa projekteissa ohjelmistokehittäjien vaihtelevista käytännöistä aiheutuneet ongelmat nousivat isompiin rooleihin. Erilaiset käytännöt aiheuttivat varsinkin ylläpidollisia haasteita komponenttien kirjoittaessa lokitiedostoja hyvinkin vaihtelevasti eri paikkoihin ja aina alkuperäinen ohjelmistokehittäjä ei ollut saatavilla tai enää edes yrityksen palveluksessa. Lisäksi komponenttien virhetilanteissa antamat virheilmoitukset olivat hyvin vaihtelevia eivätkä aina ylläpitoa auttavia. Pahimpana ongelmana oli näiden asioiden ajoittainen näkyminen myös asiakkaalle heikentäen asiakkaan omasta järjestelmästään saamaa hyötyä.

Komponenttikehyksen kehitystyön tavoitteet kumpusivat pitkälti silloisen Mepco Oy:n sisällä vallinneesta ääneen lausumattomasta tarpeesta saada aikaan jonkinlainen järjestys vallitsevaan käytäntöön, jota saattoi hieman kärjistettynä kutsua jopa anarkiaksi. Liiketoiminnasta kumpuavina tavoitteina olivat kehitystyön nopeuttaminen tarjoamalla valmis ja monistettava pohja. Yhtenäisen pohjan käytöllä saadaan ohjattua kaikille komponenteille yhteiset huolenaiheet, kuten lokitus ja virhekäsittely, samaan yhtenäiseen toimintatapaan. Tämä näkyy asiakkaan suuntaan ennenkaikkea parempana virhetilanteiden hallintana ja jäljitettävyytenä. Yrityksen sisällä

tämä vaikuttaa sekä kehityskustannuksiin että vähentää jossain määrin omistetun koodin määrää.

3. TEORIA

Diplomityössä tehdyn komponenttikehyksen suunnittelutyö vaati Dynamics CRM -alustan tuntemisen lisäksi useiden erilaisten suunnitteluperiaatteiden hyödyntämistä. Tässä luvussa on kuvattu komponenttikehyksen toteutuksen suunnittelussa hyödynnetyt ohjelmistoarkkitehtuurin ja ohjelmistokehysten teoria sekä Dynamics CRM -alustan tarpeellinen arkkitehtuuri.

3.1 Ohjelmistoarkkitehtuurista

Ohjelmiston arkkitehtuuri perustuu sen ositukseen eli dekompositioon jonkin periaatteen mukaan [27, s. 19]. Dekomposition onnistumisella on suuri merkitys muun muuassa järjestelmän eri komponenttien ylläpidettävyydelle, ymmärrettävyydelle ja uudelleenkäytettävyydelle [104, s. 1055-1056]. Tähän vaikuttaa oikein valittu dekomposition kriteeristö [104].

Erääksi hyväksi dekomposition kriteeriksi mainitaan tiedon piilottamisen periaate, jolla saadaan tuotettua löyhät riippuvuudet eri komponenttien välille [103; 104, s. 1056]. Muita dekomposition periaatteita ovat esimerkiksi funktionaalinen dekompositio sekä dekompositio tiedon abstrahoinnin pohjalta [112, s. 111]. Ohjelmiston eri huolenaiheiden erottaminen toisistaan on yksi ohjelmistotuotannon perusasioita [102, s. 1]. Ohjelmiston kyky saavuttaa sille asetetut tavoitteet riippuu pohjimmiltaan kyvystä erotella kaikki ohjelmiston tärkeät huolenaiheet toisistaan ohjelmiston kehityksen aikana [112, s. 107].

Modulaarisen dekomposition tuloksena syntyvien moduulien tulisi Meyerin [37, s. 57] mukaan olla avoimia laajentamiselle, mutta suljettuja muutoksilta. Tästä Meyerin esittämästä periaatteesta käytetään nimeä avoin-suljettu (eng. *Open-Closed principle*). Tämän periaatteen mukaan avoimuus tarkoittaa esimerkiksi mahdollisuutta lisätä moduuliin uusia kenttiä tai toimintoja. Yleisemmin ilmaistuna tämä tarkoittaa Meyerin mukaan mahdollisuutta laajentaa moduulin alkuperäistä toimintaa. Muutoksilta sulkeminen puolestaan tarkoittaa moduulin sisäisen toteutuksen suojaamista ulkoisia muutoksia vastaan [37, s. 57; 34]. Oliokielissä tätä periaatetta saadaan toteutettua esimerkiksi periyttämisellä [37, s. 60] tai rajapinta-abstraktioilla [34].

Parnaksen [104] esittämän tiedon piilottamisen periaatteen on esitetty olevan sama asia kuin avoin-suljettu -periaate [29].

Käänteinen kontrolli (eng. *Inversion of Control*) on arkkitehtuuri, jossa sovelluksen kontrolli käännetään niin, että sovellus itse ei hallitse kontrollia, vaan sovelluksen käyttämä kirjasto ottaa sovelluksen kontrollin itselleen. Tässä arkkitehtuurissa kirjaston kutsumia prosessointivaiheita voidaan räätälöidä sovelluskohtaisilla tapahtumankäsittelijöillä. Kirjasto kutsuu näitä sovelluskohtaisia tapahtumankäsittelijöitä vastineena johonkin ulkopuoliseen tapahtumaan. Yhteenvetona käänteinen kontrolli sallii sovelluksen sijasta kirjaston päättää, mitä sovelluskohtaisia toimintoja suoritetaan tietyissä tilanteissa. [14, s. 27; 12, s. 34; 24]

Riippuvuuden injektointi (eng. *Dependency Injection*) on joukko ohjelmistosuunnittelun periaatteita ja suunnittelumalleja, jotka auttavat löyhästi riippuvan ohjelmistokoodin kehittämisessä [108, s. 4]. Riippuvuuksien injektoinnin tärkein tavoite on ylläpidettävän ohjelmakoodin aikaansaaminen [108, s. 4]. Riippuvuuden injektointi on nimensä mukaisesti tarkoitettu komponentin riippuvuuksien kertomiseen (injektoimiseen) komponentille sen ulkopuolelta eli komponentin ei tarvitse itse tuntea riippuvuuksiaan [13; 108, s. 24–27]. Komponentti puolestaan kertoo rajapinnan, joka injektoitavien riippuvuuksien pitää toteuttaa [13; 108, s. 10]. Liskovin korvausperiaatteen mukaisesti tämä mahdollistaa sellaistenkin riippuvuuksien injektoinnin, joita ei oltu alunperin edes ajateltu, kunhan riippuvuus vain toteuttaa vaaditun rajapinnan [13; 108, s. 10-11]. Liskovin korvausperiaate (eng. *Liskov substitution principle*) on olio-ohjelmoinnin periaate, jossa viitattu kantaluokka voidaan korvata millä tahansa siitä periytetyn lapsiluokan toteutuksella ilman tarvetta muuttaa kantaluokasta riippuvaa toteutusta [30; 33].

Riippuvuuksien injektointia voidaan tehdä useilla eri tavoilla. Tapoja ovat esimerkiksi rakentajan tai metodin kautta injektointi. Rakentajan kautta injektoitavia asioita ovat tyypillisesti sellaiset riippuvuudet, joita ilman luokan instanssi ei pysty lainkaan toimimaan. Näistä annetuista viittauksista säilötään riippuvuudet luokan jäsenmuuttujina. Rakentajan kautta injektointia kehotetaan käyttämään ensisijaisena injektointitapana, jos muuta perustetta ei ole. Metodin kautta injektointia puolestaan käytetään riippuvuuden vaihdellessa metodin jokaisella suorituskerralla. [108]

Eräs ohjelmistojen helppoon ylläpidettävyyteen liittyvä periaate on käänteisten riippuvuuksien periaate (eng. *Dependency Inversion Principle*). Tämä periaate juontuu avoin-suljettu -periaatteen ja Liskovin korvausperiaatteen aiheuttamista ohjelmiston rakenteista. Periaatteen mukaan korkean tason moduulien ei tule riippua matalan tason moduuleista, vaan molempien on riiputtava abstraktioista. Lisäksi abstrak-

tioiden ei pidä riippua yksityiskohdista, vaan yksityiskohtien tulee riippua abstraktioista. Periaatteen mukaisissa toteutuksissa abstraktio eristetään käsiteltävän aiheen yksityiskohdista, minkä jälkeen toteutuksen yksityiskohdat ohjataan riippumaan kyseisestä abstraktiosta. Näin pyritään aikaansaamaan sellainen ohjelmisto, joka on helposti muuteltavissa, jossa yksittäiset koodimuutokset eivät aiheuta heijastusvaikutuksia muualle ohjelmistoon ja joka on helposti uudelleenkäytettävissä. Tämän periaatteen noudattamisen sanotaan olevan välttämätöntä uudelleenkäytettävien ohjelmistokehysten luontiin. [32]

Komponentilla voi olla useita eri rooleja suhteessa muihin komponentteihin [27, s. 77]. Tällöin komponentti toteuttaa useita eri roolirajapintoja [27, s. 77]. Roolirajapinta kuvaa yhden selkeästi identifioidun roolin, jossa komponentti palvelee toista komponenttia [27, s. 77]. Rooliperusteisiin rajapintoihin pohjautuvasta järjestelmästä tulee hienojakoinen ja siten helpommin ylläpidettävä [27, s. 79]. Käänteisen kontrollin periaatteen [12] voidaan ajatella käyttävän abstraktioiden eristysmekanismina esimerkiksi roolirajapintoja. Niin käänteinen kontrolli [12], riippuvuuksien injektointi [108], käänteisten riippuvuuksien periaate [32] kuin roolirajapinnatkin [27] pyrkivät kaikki parantamaan ohjelmiston ylläpidettävyyttä. Näiden kaikkien toteutusmekanismit ovat melko samankaltaisia. Tästä huolimatta näillä periaatteilla on keskenään selkeitä eroja, minkä takia ne on esitelty toisistaan erillisinä periaatteina.

Joskus on tarvetta käyttää ohjelmiston useaa eri moduulia tai rajapintaa yksinkertaisen rajapinnan läpi. Fasadi (eng. *Facade*) on suunnittelumalli, joka määrittelee korkeamman abstraktiotason rajapinnan helpottamaan eri moduulien yhteiskäyttöä. Fasadia hyödyntämällä saadaan yksinkertaisempia rajapintoja näiden moduulien käyttöön. Fasadin avulla pystytään lisäksi edistämään löyhästi riippuvan ohjelmistokoodin toteuttamista ja toteuksen jakamista eri kerroksiin. [14, s. 185–188]

Kun haluaan tehdä komponentti riippumattomaksi sen vaatimien muiden komponenttien toteutuksista, yleisin ratkaisu on käyttää rajapintoja [27, s. 76]. Gamma *et al.* [14] esittävät silta-suunnittelumallia (eng. *Bridge*) mahdollistamaan komponentin riippuvuuksien ja näiden toteutusten keskinäinen riippumattomuus. Esitetty suunnittelumalli kuvaa, miten rajapinnan avulla voidaan tarpeesta riippuen käyttää eri luokkien toteutuksia ja vaihtaa niitä ilman vaikutuksia rajapinnan kutsujan toteutukseen. Lisäksi kuvataan, miten suunnittelumallin käyttö piilottaa toteutuksen yksityiskohdat sekä mahdollistaa myös rajapintojen periyttämisen ja yhdistelemisen.

Kun käytetään olioita, joiden luonti on kallis operaatio tai tarvittavien olioiden lukumäärä on rajallinen, jo luotuja olioita voidaan käyttää uudestaan. Olioallas (eng. *Object Pool*) on tällaiseen käyttöön tarkoitettu suunnittelumalli. Tässä suunnittelu-

mallissa jo luotuja olioita pidetään säilöttynä johonkin tietorakenteeseen. Jo luodut oliot ovat keskenään vaihtokelpoisia. Vaihtokelpoisuuden takaamiseksi jokaisen tietorakenteessa tallessa olevan olion tilan on oltava tunnettu ja sama. [19, s. 159-174]

Tyypitetyissä kielissä voi koodin kirjoittamisen aikana tulla vastaan tilanteita, joissa tietorakennetta tai algoritmia suunniteltaessa kaikki ajonaikaisesti käytettävät tietotyypit eivät vielä ole tiedossa [106, s. 231–232; 31]. Esimerkiksi C++-kielessä tämä parametroitavien tyyppien toteutustekniikka tunnetaan englanninkielisellä nimellä *template*, joka voidaan suomentaa malliksi [106, s. 232]. Malleissa jätetään avoimeksi tiettyjen, tyyppiparametreiksi kutsuttujen parametrien tietotyypit [106, s. 232]. Mallia käyttävässä ohjelmakoodissa määritetään tyyppiparametrien tietotyypit mallin määräämien rajoitteiden puitteissa [106, s. 236–237]. Mallit voivat olla joko funktiomalleja (eng. *function template*) tai luokkamalleja (eng. *class template*) [106, s. 232]. Mallien avulla saadaan vältettyä tietyissä tilanteissa esimerkiksi runsasta lapsiluokkien toteuttamistarvetta [14]. Yleistetymmin malleilla saadaan toteutettua geneerisiä, tyyppiriippumattomia algoritmeja [106, s. 249; 31]. .NET-kielissä, kuten C#-kielessä, *template*-konseptia vastaavasta toteutusmekanismista käytetään englanninkielistä termiä *generics* [31]. Ominaisuuksien ja toteutuksen puolesta nämä kaksi eroavat merkittävästi toisistaan [31], mutta tämän diplomityön näkökulmasta näillä eroilla ei ole merkitystä.

Joidenkin järjestelmien pääasiallinen tehtävä on tietovirran jalostamista ja prosessointia. Tällaiseen tarpeeseen sopii tietovuoarkkitehtuuri (eng. *pipes-and-filters architecture*). Tässä arkkitehtuurissa käytetään väyliä (*pipe*) tietovirran kuljettamiseen tiedon prosessointiyksiköiden (*filter*) välillä. Näistä jokainen prosessointiyksikkö lukee omaa syötevirtaansa ja tuottaa prosessoinnin jälkeen oman tulostevirtansa. Prosessointiyksiköt toimivat toisistaan riippumattomina ja prosessoitavien tietoalkioiden ei tulisi riippua toisistaan. Yksinkertaisin tietovuoarkkitehtuurin muoto on liukuhihna-arkkitehtuuri (eng. *pipeline architecture*). Tässä tietovirta ei haaraudu, vaan etenee peräkkäin tiettyä prosessointiyksiköiden ketjua pitkin. [27, s. 132–133]

Muuttujan arvo, luokan tyyppi tai kutsuttava metodi voidaan sitoa ennen ohjelman suoritusta niin, että se ei muutu ajonaikaisesti [111, s. 346]. Vaihtoehtoisesti vastaava sidonta voidaan tehdä vasta ohjelman suorituksen aikana [111, s. 116]. Ennen ohjelman suoritusta tehtävää sidontaa kutsutaan staattiseksi sidonnaksi [111, s. 346]. Ohjelman ajonaikaista sidontaa kutsutaan dynaamiseksi sitomiseksi [111, s. 116; 106, s. 133]. Yksi staattisen sidonnan hyöty on joidenkin ohjelman virheiden aikaisempi havaitseminen ohjelmiston kehityksen aikana [2; 50]. Vaikka ensisijaisesti voidaankin suositella staattista sidontaa, dynaamisen sidonnan käyttäminen on paikoin oikea vaihtoehto [2; 90]. Tämän diplomityön osalta tärkein staattisen ja dynaami-

sen sidonnan esiintymismuoto on kohdassa 3.4 tarkemmin kuvattu käännösaikainen tyypityksen tarkistaminen tietyissä Dynamics CRM -järjestelmän ohjelmointitöissä.

3.2 Ei-toiminnallisten vaatimusten erottelu

Järjestelmällä voi olla toiminnallisia ja ei-toiminnallisia vaatimuksia, joihin sen arkkitehtuurin on vastattava [9, s. 91]. Toiminnallinen vaatimus määrittelee sellaisen toiminnon, joka järjestelmän on kyettävä toteuttamaan [111, s. 153]. Ei-toiminnallisen vaatimuksen määritelmästä ei puolestaan ole olemassa yhtenäistä näkemystä, kuten Chung [8] ja Glinz [17] toteavat. Ei-toiminnallinen vaatimus ei kuitenkaan kuvaa sitä, mitä ohjelmiston pitää tehdä [17; 111, s. 231] Ei-toiminnallinen vaatimus kuvaa miten ohjelmisto tekee sen mitä se tekee [111, s. 231]. Ei-toiminnalliset vaatimukset tyypillisesti asettava ohjelmistolle erilaisia vaatimuksia [8; 17] tai rajoitteita [17]. On myös esitetty, että valtaosassa järjestelmään tehdyistä arkkitehtuurisista valinnoista on taustalla pyrkimys saavuttaa tietyt laadulliset tai ei-toiminnalliset vaatimukset [1; 22, s. 115; 115, s. 1].

Ohjelmiston kehittämisen eri näkökulmat ja ongelmat ovat usein osittain päällekkäisiä ja vuorovaikutuksessa toistensa kanssa [112]. Ohjelmiston kaikkia oleellisia asioita on hankala kuvata ja toteuttaa sekä olio-ohjelmoinnin että proseduraalisen ohjelmoinnin avulla [25, s. 220]. Syynä kuvaamisen hankaluuteen on näiden ongelmien läpileikkaavuus (eng. *cross-cut*) järjestelmän perustoimintojen läpi [25, s. 220]. Järjestelmän perustoimintojen läpi leikkaavia ongelmia kutsutaan järjestelmän aspekteiksi [25, s. 221]. Erilaiset aspektit liittyvät usein ohjelmiston ei-toiminnallisiin vaatimuksiin [25, s. 226–227]. Esimerkkejä eri sovellusalueilla yleisistä vastaan tulevista aspekteista ovat muun muuassa virhekäsittely ja tiedon tallennus [25, s. 227; 112, s. 111]. Ohjelmiston aspektien toisistaan erottamisen edut tunnetaan hyvin ja nykyaikaisten kehitysvälineet tarjoavat tukea aspektien erottamiseen [112, s. 107]. Tästä huolimatta ohjelmistoissa näkyy niiden elinkaaren aikana haitallisia ominaisuuksia, jotka tyypillisesti yhdistetään puutteelliseen aspektien erotteluun [102, s. 1]

Jos läpileikkaavan ongelman ratkaisussa tukeudutaan puhtaasti käytössä olevan ohjelmointikielen kompositiomekanismiin, saadaan aikaan monimutkaista ja sekavaa lähdekoodia [25, s. 226]. Tarr *et al.* [112, s. 109] spekuloivat tämän syyksi sitä, että olemassa olevat menetelmät tukevat yleensä vain yhtä samanaikaista dekompositiotapaa, jolloin tästä kyseisestä tavasta tulee hallitseva ja muut samanaikaiset aspektit jäävät tämän hallitsevan dekomposition alle. Voidaankin todeta, että jos järjestelmän läpileikkaavaa ongelmaa ei saada kuvattua yhtenä moduulina, ratkaisun hyödynnettävyys ja uudelleenkäytettävyys mitä todennäköisimmin kärsivät toteutuksen hajautuessa monen eri operaation yli [4, s. 51].

3.3 Ohjelmistokehykset

Ohjelmistokehys on sellainen ohjelmistojen uudelleenkäyttöä tukeva ohjelmistotuote, josta voidaan erikoistamalla tuottaa räätälöityjä sovelluksia [24; 27, s. 187]. Ohjelmistokehys ei tyypillisesti ole suorituskelpoinen ohjelmisto, vaan sitä pitää täydentää ennalta oletetuista kohdista [14, s. 26; 27]. Ohjelmistokehys kapseloi konkreettiset toteutukset vakaiden rajapintojen taakse [12, s. 32]. Lisäksi ohjelmistokehys yleisemmin kuvaa käytettävät sovellusalueen suunnitteluratkaisut [14, s. 27]. Tämä modulaarisuus auttaa sekä parantamaan sovelluksen laatua että helpottamaan sen ylläpitoa [14, s. 27; 12, s. 32–34]. Ohjelmistokehyksen uudelleenkäytettävyyden kantavana ideana on hyödyntää aiemmin tehtyä työtä sekä olemassa olevaa osaamista joltain tietyltä osaamisalueelta [12, s. 34]. Ohjelmistokehyksen erikoistamisen tuloksena voidaan saada joko itsenäinen sovellus tai komponentti [27]. Näistä jälkimmäisessä tapauksessa ohjelmistokehystä kutsutaan komponenttikehykseksi.

Ohjelmistokehykseen on sen toteutuksen aikana jätetty tarkoituksellisia aukkoja, joihin voidaan kiinnittää räätälöityä sovelluskohtaista ohjelmakoodia. Näitä aukkoja kutsutaan laajennuskohdiksi. Ohjelmistokehys asettaa vaatimukset siitä, millaista sovelluskohtaista ohjelmakoodia näihin laajennuskohtiin voidaan kiinnittää. Ohjelmistokehyksen erikoistamisrajapinta tarkoittaa sekä sen laajennuskohtia että niitä täyttävälle sovelluskohtaiselle ohjelmakoodille asetettuja vaatimuksia. [14, s. 26–27; 27, s. 187–189]

Perinteisesti kirjastojen uudelleenkäytössä suorituksen pääkontrolli on sovelluksella, josta kutsutaan tarvittavissa kohdissa kirjaston palveluja [27]. Yksi ohjelmistokehyksien tunnusomainen piirre on erikoistetun, sovelluskohtaisen ohjelmakoodin kutsuminen kehyksen sisältä takaisinkutsujen avulla eikä suoraan varsinaisen sovelluksen ohjelmakoodista [24; 27, s. 191]. Ohjelmistokehys toimiikin usein koko sovelluksen pääohjelmana, koordinoiden sovelluksen suoritusta ja kutsuen tietyissä kohdissa sovellusta varten erikoistettua ohjelmakoodia käänteisen kontrollin periaatteen mukaisesti. [24; 14, s. 26–27]

Ohjelmistokehykset voidaan luokitella käyttämiensä laajennustekniikoiden pohjalta muunneltaviin kehyksiin (eng. *white-box framework*) ja koottaviin kehyksiin (eng. *black-box framework*) sekä näiden välimuotoihin [12, s. 35]. Muunneltavat kehykset tukeutuvat tyypillisesti oliokielten mekanismeihin, kuten periyttämiseen ja dynaamiseen sitomiseen [12, s. 35]. Muunneltavien kehysten käyttöönotto on tyypillisesti hankalaa, koska se vaatii samalla opettelemaan, miten kyseinen kehys on toteutettu [24]. Koottavat kehykset puolestaan ovat räätälöitävissä erikoistamisrajapintojen kautta [24]. Tällöin kehykselle annetaan käyttäjän toimesta joukko sovelluskohtaisia

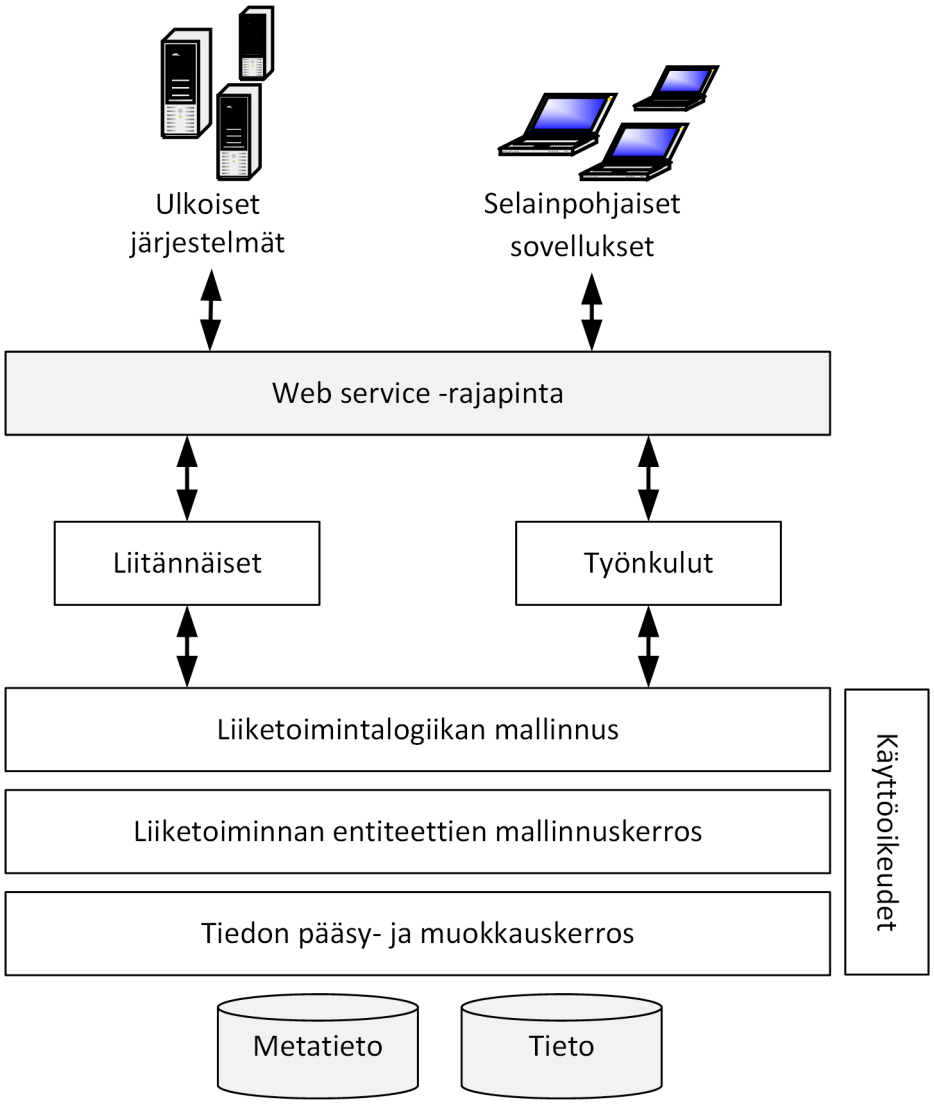
komponentteja, jotka toteuttavat nämä erikoistamisrajapinnat [24]. Tämä on käyttöönottajaystävällisempi tapa, sillä käyttäjän pitää ymmärtää vain erikoistamisrajapinta eikä myös sisäistä toteutusta [24]. Tyypillisesti koottavat kehykset ovat helpompia käyttää ja laajentaa kuin muunneltavat kehykset [12, s. 35]. Lisäksi, koottavien kehyksien tulisi ajan myötä korvata muunneltavat kehykset, kun ymmärrys sovelluksen toiminnasta ja sovelluksen asettamista vaatimuksista ohjelmistokehykselle kasvaa [24].

3.4 Dynamics CRM

Dynamics CRM on yritysten liiketoimintasovellusten rakentamiseen tarkoitettu alusta [94]. Tällainen alusta mahdollistaa liiketoimintaa tukevien sovellusten kehitystyön ilman, että tarvitsee ensin käyttää aikaa liiketoimintasovelluksen alustan kehitystyöhön. Dynamics CRM on syntyjään asiakkuudenhallinnan järjestelmä, mutta sen käyttömahdollisuudet eivät rajoitu vain asiakkuudenhallintaan. Dynamics CRM -järjestelmän laajentamisen ydin on sen tehokas laajennettavuusmalli [78]. Tämä tarkoittaa rakennettavan liiketoimintajärjestelmän metatietojen ja prosessien kuvaamista käyttämällä siihen tarjolla olevia Dynamics CRM -työkaluja. Kuvattavat prosessit voivat sisältää alustan omilla työkaluilla tehtäviä työnkulkuja tai ohjelmoimalla tehtäviä liitännäisiä. Työnkulkuja voidaan laajentaa ohjelmoinnin avulla sisällyttämällä niihin räätälöityjä työnkulun vaiheita [48]. Dynamics CRM -alusta ei pakota toimimaan minkään tietyn liiketoimintaprosessin mukaan, kun järjestelmää rakennetaan [78]. Dynamics CRM -alusta tarjoaa pääsyn sen sisälle tallennettuun tietoon. Kaikki tiedon käsittely tapahtuu Dynamics CRM -alustan web service -rajapinnan kautta. Tähän on poikkeuksena tietty raportointiskenaario, jossa tiedot voidaan lukea SQL-tietokannasta. Tämän diplomityön näkökulmasta Dynamics CRM voidaan käsittää alustana, jossa kaikki vuorovaikutus järjestelmään tallennetun tiedon kanssa tehdään web service -rajapinnan läpi.

Dynamics CRM -järjestelmä käyttää metatietopohjaista arkkitehtuuria tietomallin rakentamiseen [79]. Metatietoa hyödynnetään tietomallin ylläpidon lisäksi esimerkiksi Dynamics CRM:n käyttöliittymän rakentamisessa. Tietomallia kuvaava metatieto sisältää tiedot muun muuassa entiteeteistä, entiteettien kentistä ja entiteettien välisistä relaatioista. Entiteetti on tietosäiliö, johon tallennetaan Dynamics CRM -järjestelmän käsittelemä tieto. Entiteetti koostuu sen kentistä. Yksi kenttä on tarkoitettu yhden yksittäisen tiedon tallentamiseen. Konseptimielessä entiteetin voi ajatella olevan sama asia kuin SQL-tietokannassa oleva yksittäinen tietokantataulu [60]. Entiteetin kentän puolestaan voi mieltää olevan sama asia kuin SQL-tietokannassa olevan taulun sarake. Entiteettiin tallennettu entiteetin yksi ilmentymä on tietue. Tietue

vastaa konseptimielessä samaa asiaa kuin yksi rivi SQL-tietokannan taulussa. Esimerkiksi *Contact*-entiteetissä voi olla kentät etunimi, sukunimi sekä pääavain. Esimerkki tähän entiteettiin lisättävästä tietueesta voisi olla seuraava: etunimi-kenttä saa arvon "Teemu", sukunimi-kenttä saa arvon "Teekkari" ja pääavain numeroidaan automaattisesti. Tämän jälkeen *Contact*-entiteetissä on yksi tietue, jonka tietosisältö on edellä kuvattu. Dynamics CRM:n laajennusarkkitehtuurin eri osia havainnollistaa kuva 3.1.



Kuva 3.1 *Dynamics CRM -järjestelmän laajennusarkkitehtuuri mukailtuna lähteestä [78]. Kuvasta nähdään järjestelmän laajennusmahdollisuudet. Harmaalla pohjavärillä korostettuja osia ei pystytä muokkaamaan. Muita järjestelmän osia pystytään muokkaamaan.*

Dynamics CRM on myös tapahtumapohjainen järjestelmä [54]. Kaikki Dynamics CRM -järjestelmää käyttävän loppukäyttäjän tekemät toimenpiteet aiheuttavat kutsun lähettämisen alusta web service -rajapintaan. Tämä tarkoittaa sekä järjestelmän oman käyttöliittymän käyttöä että ulkopuolisen ohjelman tekemää suoraa kutsua

web service -rajapintaan [78]. Järjestelmän oman käyttöliittymän käyttö aiheuttaa sisäisen web service -rajapinnan kutsun. Jokainen web service -kutsu aiheuttaa Dynamics CRM:n sisällä uuden yksittäisen tapahtuman [54]. Jokainen tällainen tapahtuma siirtyy Dynamics CRM:n tapahtumankäsittelylinjaston käsiteltäväksi. Tämän diplomityön puitteissa Dynamics CRM:n tapahtumankäsittelylinjasto voidaan olettaa liukuhihna-arkkitehtuuriksi.

Tapahtumankäsittelylinjasto voi käsitellä tapahtumia sekä synkronisesti että asynkronisesti [54]. Synkroninen käsittely tapahtuu välittömästi tapahtuman yhteydessä. Synkronisen käsittelyn aikana linjaston sisäinen käsittely etenee tietyssä hyvin määritellyssä järjestyksessä. Synkroninen käsittely tehdään aikakriittisille liitännäisille ja työnkuluille. Asynkroninen käsittely tehdään niille liitännäisille ja työnkuluille, jotka eivät ole aikakriittisiä [39]. Synkronisen käsittelyn lopuksi tapahtumankäsittelylinjasto lisää nämä ei-aikakriittiset työnkulut ja liitännäiset asynkronisten tapahtumien käsittelyjonoon [54]. Asynkroniset tapahtumat käsitellään myöhemmin erillisen prosessointipalvelun toimesta. Tämän diplomityön näkökulmasta tapahtumankäsittelylinjasto on merkittävä asia, koska tapahtumankäsittelylinjasto liittyy hyvin tiiviisti varsinkin liitännäisten kehitystyöhön ja liitännäiset ovat diplomityössä toteutettavan komponenttikehyksen erikoistuksen lopputuloksia. Synkronisen ja asynkronisen tapahtumankäsittelyn keskinäisillä eroilla ei kuitenkaan ole merkittävää vaikutusta diplomityöhön.

Tapahtumankäsittelylinjasto on luonteeltaan rekursiivinen [58]. Suurinta sallittua rekursion syvyyttä on kuitenkin rajoitettu järjestelmän suojelemiseksi esimerkiksi ohjelmointivirheen takia tehdyltä ikuiselta silmukalta. Rekursiolla tarkoitetaan sitä, että tapahtumankäsittelylinjastolla ajettava liitännäinen tai työnkulku käynnistää toiminnallaan uuden tapahtuman käsittelyn. Käytännössä tämä tehdään kutsumalla liitännäisestä tai työnkulusta Dynamics CRM:n web service -rajapintaa. Käynnistetty uusi tapahtuma linkittyy alkuperäiseen tapahtumaan. Alusta tunnistaa toisiinsa liittyvät tapahtumat antamalla niille kaikille saman alkuperäisestä tapahtumasta perityn yksilöllisen korrelaatiotunnuksen [57].

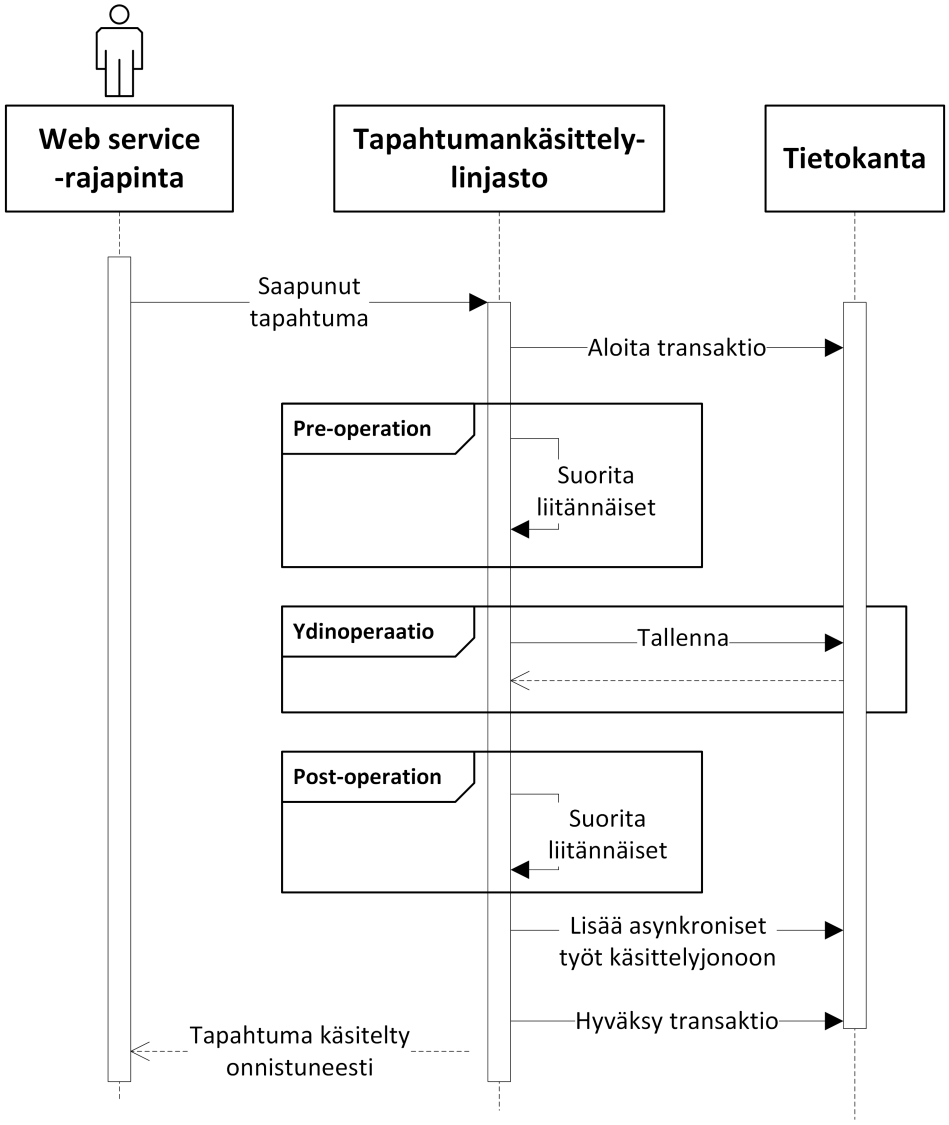
Tapahtumankäsittelylinjastolla tapahtuvassa prosessoinnissa on neljä eri tilaa [54], joista seuraavat kolme ovat tämän diplomityön kannalta merkityksellisiä. Nämä kolme vaihetta muodostavat yhdessä kriittisen alueen ja niiden yli tapahtuu poissulkeminen kyseisessä transaktiossa osallisina oleviin tietueisiin [16]. Tapahtumankäsittelylinjaston toimintaa näiden tilojen yhteydessä havainnollistaa kuva 3.2. Kuvasta nähdään, miten tapahtumankäsittelylinjaston kriittisen alueen sisällä olevat kolme eri tilaa, transaktion laajuus ja asynkronisten töiden ajastus suhtautuvat keskenään.

- *Pre-operation*, joka tarkoittaa tietokantatransaktion sisällä olemista kyseisen tapahtuman osalta, mutta tapahtumassa kulkevan tietueen muutoksia ei ole vielä tallennettu tietokantaan [54].
- *MainOperation*, joka tarkoittaa Dynamics CRM -järjestelmän sisäistä tilaa, jonka aikana suoritetaan järjestelmän ydinoperaatio, kuten tietueen luonti, päivitys tai poisto, eli tapahtumassa kulkevat tietueen muutokset tallennetaan tässä yhteydessä tietokantaan [54]. Tämän vaiheen tarkoitus on taata yhtenevästi tapahtuvat tietokantaoperaatiot huolimatta mahdollisista transaktiossa osallisena olevista liitännäisistä [16]. Tässä tilassa tapahtuvaa operaatiota kutsutaan myöhemmin suomenkielisellä termillä ydinoperaatio.
- *Post-operation* on sellainen tietokantatransaktion sisällä oleva vaihe, joka suoritetaan ydinoperaation jälkeen [54].

Liitännäiset ja työnkulut rekisteröidään käynnistymään haluttujen tapahtumankäsittelylinjaston tapahtumien yhteydessä [54; 88]. Rekisteröinnin yhteydessä otetaan kantaa siihen, missä tapahtumankäsittelylinjaston prosessointivaiheessa liitännäinen tai työnkulku ajetaan. Yhdestä tapahtumasta voi käynnistyä monta liitännäistä tai työnkulkua. Kun tapahtumankäsittelylinjasto käsittelee tapahtumaa, käsittelyyn liittyy suorituskonteksti [54]. Suorituskonteksti välitetään automaattisesti liitännäisille ja räätälöidyille työnkulun vaiheille [44; 54]. Suorituskonteksti sisältää tapahtumaan liittyvää ajonaikaista suoritusympäristöä kuvaavaa tietoa [80]. Tapahtumankäsittelylinjasto luo suorituskontekstin automaattisesti jokaiselle käsittelemälleen tapahtumalle.

Suorituskontekstin sisältämiä tietoja ovat muun muuassa tieto käsittelyn käynnistäneestä tapahtumasta, tapahtumaan liittyvä tietue ja tapahtuman käsittelyyn liittyvä ulkopuolinen syöte [80]. Suorituskonteksti välitetään tapahtumasta käynnistyvälle liitännäiselle tai työnkululle automaattisesti tietyn Dynamics CRM -rajapinnan instanssina[1]. Käsittelyn käynnistänyt tapahtuma tunnistetaan tämän rajapinnan *MessageName*-kentän sisällöstä [68]. Esimerkiksi muokkaustapahtuman yhteydessä tässä kentässä on arvo “Update”. Muokkaustapahtuman yhteydessä suorituskonteksti viittaa tapahtumassa käsiteltävään tietueeseen. Viittaus tehdään käyttämällä kyseisen entiteetin nimeä ja tietueen pääavainta. Entiteetin nimi voi olla esimerkiksi *Contact*. Muokkaustapahtuman yhteydessä saatava ulkopuolinen syöte on loppukäyttäjän muuttama tietueen tietosisältö [80]. Loppukäyttäjä voi olla korjannut henkilön etunimessä olleen kirjoitusvirheen muuttamalla tietueen etunimikentässä

[1]Kuvaus siitä, miten suorituskonteksti teknisesti välitetään, on tämän kohdan sivulla 25.



Kuva 3.2 *Dynamics CRM -järjestelmän tapahtumankäsittelylinjasto UML:n sekvenssikaaviona mukailtuna lähteestä [54]. Kuvasta nähdään miten tapahtumankäsittelylinjasto käsittelee web service -rajapinnan kautta saapunutta tapahtumaa.*

olleen arvon "Teem" arvoon "Teemu". Tällöin tapahtuman saama ulkopuolinen syöte olisi etunimikenttä, jonka sisällä on arvo "Teemu". Tämä etunimikentän arvo on tässä tapauksessa myös ainoa ulkopuolinen syöte. Suorituskontekstin sisältämät tiedot antavat liitännäistä toteuttavalle ohjelmistokehittäjälle paljon tarpeellista tietoa asiakaskohtaisen logiikan toteuttamiseen [89]. Ohjelmistokehittäjä voi muuttaa järjestelmän ydinoperaatiolle välitettävää tietosisältöä muokkaamalla suorituskontekstin sisältöä [80].

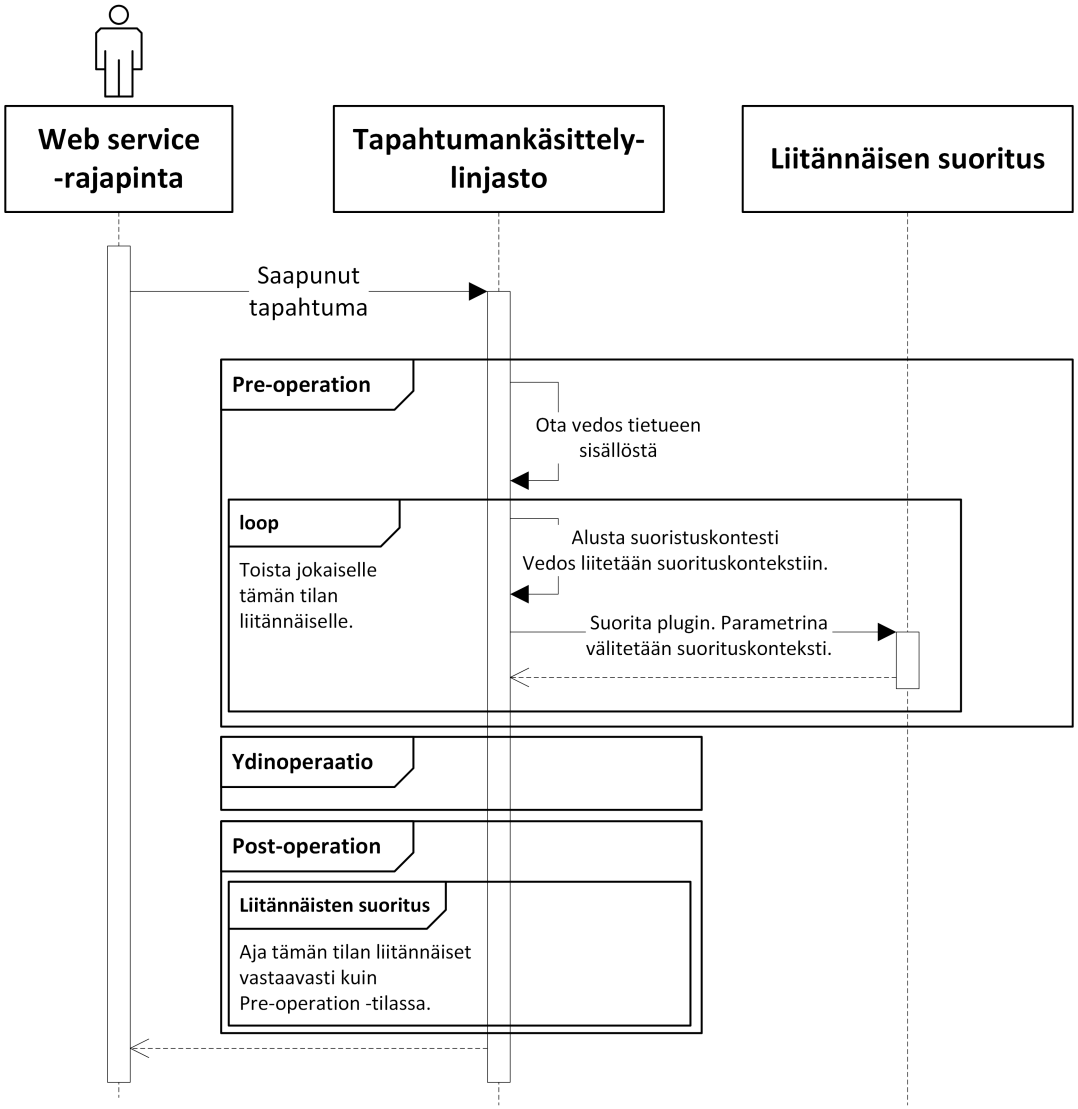
Suorituskonteksti voi sisältää muitakin tietoja, kuten vedoksia (eng. *snapshot)*. Vedos kuvaa tapahtumankäsittelylinjaston käsittelemän yksittäisen tietueen tietosisältöä jollakin tietyllä ajanhetkellä. Vedos sisältää käsiteltävän tietueen kenttien ar-

vot sellaisina kuin ne olivat kyseisellä ajanhetkellä. Vedoksessa ydinkonsepti on se, miltä yksi yksittäinen tietue on näyttänyt juuri jonakin tiettynä hyvin yksilöitynä ajanhetkenä. Vedos ei kuitenkaan ole tietue sen lopullisessa, transaktion jälkeisessä muodossaan. Vedoksen ottamisen ajankohdan absoluuttisella arvolla ei ole merkitystä, vaan sillä, miten se suhtautuu tapahtumankäsittelylinjaston aiemmin mainittuihin tiloihin. Muutamaa poikkeusta lukuunottamatta vedoksia voidaan pyytää tapahtuman suorituksen alusta tai välittömästi ydinoperaation jälkeen. Vedoksia ei voi ottaa esimerkiksi tietueen sisällöstä ennen luontitapahtumaa, koska tietuetta ei ole ollut olemassa tietokannassa ennen luontitapahtuman ydinoperaatiota. Tapahtuman käsittelyn alussa otettu vedos kuvaa tietueen tietosisällön, johon mahdollinen transaktion peruuttaminen palaa. Ydinoperaation jälkeen otettu vedos puolestaan kertoo tietueen tietosisällön välittömästi ydinoperaation jälkeen ja ennen kuin yksikään tapahtuman yhteydessä käynnistyvä liitännäinen tai työnkulku on käynnistynyt. Käytännössä jälkimmäinen vedos on tietue sellaisenaan kuin se tallennettiin tietokantaan. Näistä vedostyypeistä käytetään dokumentaatiossa vastaavasti termejä *PreImage* ja *PostImage*. Näitä vedoksia voidaan pyytää Dynamics CRM:n suorituskontekstin rajapinnan kautta. Vedoksen yksilöinti tehdään ohjelmistokehittäjän toimesta antamalla vedokselle haluttu uniikki tunniste. Esimerkiksi kun *Contact*-entiteetin yhden tietueen etunimikenttä muutetaan arvosta “Teem” arvoon “Teemu”, *PreImage*-vedos sisältäisi etunimikentän arvolla “Teem”. [80]

Suorituskontekstin, vedosten ja tapahtumankäsittelylinjaston eri tilojen yhteyttä havainnollistaa kuva 3.3. Kuva esittää yhden tapahtumankäsittelylinjaston tapahtuman käsittelyä. Kuvasta on jätetty pois joitakin kuvassa 3.2 esitettyjä asioita, kuten transaktio ja asynkroniset työt. Kuvasta nähdään, miten tapahtumankäsittelylinjaston kriittisen alueen sisällä olevat kolme eri tilaa[2] suhteutuvat niiden sisällä suoritettaviin liitännäisiin ja vedoksiin. Todelliseen Dynamics CRM -järjestelmään voidaan rekisteröidä useita liitännäisiä samaan tapahtumankäsittelylinjaston tapahtumaan ja tilaan. Tällöin liitännäiset suoritetaan peräkkäin yksi toisensa jälkeen tapahtumankäsittelylinjaston tilan (*pre-operation* tai *post-operation*) muuttumatta. Linjastolla aiemmin suoritetut liitännäiset pystyvät vaikuttamaan linjastolla myöhemmin suoritettavien liitännäisten saamaan suorituskontekstiin. Tyypillinen suorituskontekstin muuntelun käyttötapaus on lisätä tallennettavaan tietueeseen tietoa ennen kuin tietue tallennetaan ydinoperaatiossa tietokantaan.

Tämän kappaleen tiedot pohjautuvat pääosin käytännön kautta saatuihin kokemuksiin Dynamics CRM -alustan toiminnasta. Näitä asioita ei ole suoraan mainittu missään dokumentaatiosta. Vedoksien käyttäjä on liitännäistä toteuttava ohjelmis-

[2]Katso tapahtumankäsittelylinjaston tilat sivulta 15.



Kuva 3.3 *Yksinkertaistettu esitys Dynamics CRM -järjestelmän tapahtumankäsittelylinjaston kontekstien, liitännäisten ja vedosten keskinäisistä suhteista UML:n sekvenssikaaviona. Kaavio on mukailtu lähteistä [54] ja [80]. Tietueen sisältö poimitaan vedokseen sellaisena, kuin se on tilan suorituksen alkaessa. Suorituskonteksti alustetaan jokaisen liitännäisen suorituksen alkaessa.*

tokehittäjä. Vedoksista saadaan pääteltyä tietoja, jotka ovat joskus oleellisia toteutettavan liiketoimintalogiikan kannalta. Esimerkiksi *PreImage*-vedosta ja suorituskontekstista poimittavaa tapahtuman käynnistänyttä syötettä vertaamalla saadaan pääteltyä, mitkä tietueen kentät muuttuivat ja miten ne muuttuivat. Tämä mahdollistaa muun muuassa asiakkaan liiketoiminnan kieltämien siirtymien estämisen. Toinen syy vedosten käyttöön on suorituskyky: Dynamics CRM -alustan suorituskyvyn kannalta on tehokkaampaa pyytää tietueen tietosisältö vedoksena kuin hakea se web service -rajapinnan läpi erillisellä kyselyllä [80]. Yksi tyypillinen *PostImage*-vedoksen käyttökohde on tietueen tietosisällön selvittäminen ydinoperaation jälkeen. Tapahtumankäsittelylinjasto saattaa suorittaa useita liitännäisiä ennen kysei-

sen tapahtuman ydinoperaatiota. Jokainen näistä liitännäisistä voi muuttaa ydinoperaatiolle menevää tietosisältöä. *PostImage*-vedos on varma tapa saada käsiteltävän tietueen tietosisältö käsiteltäväksi ohjelmakoodissa sellaisena kuin tietue on tietokantaan tallennuksen jälkeen.

Liitännäisten ja räätälöityjen työnkulkujen toteutusten asennus aloitetaan tallentamalla toteutuksen sisältävä kirjasto joko Dynamics CRM:n tietokantaan tai Dynamics CRM:n asennuspalvelimen kovalevylle [93]. Dynamics CRM -alustaan voidaan rekisteröidä liitännäisiä ja räätälöityjä työnkulun aktiviteetteja suoritettavaksi sekä hiekkalaatikon sisällä että sen ulkopuolella [72]. Hiekkalaatikkoon rekisteröinnin yhteydessä kirjasto pitää tallentaa aina tietokannan sisälle [93]. Rekisteröinti hiekkalaatikon ulkopuolella suoritettavaksi ei ole mahdollista Dynamics CRM:n pilviversiossa [72]. Hiekkalaatikon sisäpuolelle rekisteröitäessä liitännäisten ja räätälöityjen työnkulun vaiheiden oikeuksia on rajoitettu [72]. Kattavaa listaa rajoitetuista oikeuksista ei ole tarjolla, mutta listattujen rajoitusten joukossa on muun muuassa estetty pääsy käyttöjärjestelmän levyjärjestelmään ja rekisteriin sekä joidenkin verkkoprotokollien käytön rajoitus [72].

Suorituskykysyistä Dynamics CRM -alusta pitää jo luotuja liitännäisolioita palvelimen välimuistissa jonkin olioaltaan tyyppisen, mutta määrittelemättömän arkkitehtuuriratkaisun avulla [89]. Kaikkien Dynamics CRM -liitännäisten toteutusten tulee olla tämän takia tilattomia eikä toteutuksessa voi luottaa liitännäisen rakentajan tulevan ajetuksi joka suorituskerralla [89]. Suorituskykysyistä liitännäiset suositellaan toteuttamaan sellaisiksi, joiden suoritus kestää mahdollisimman lyhyen aikaa [40].

Dynamics CRM -alusta mahdollistaa virheilmoitustekstien näyttämisen loppukäyttäjälle liitännäisestä käsin [56]. Loppukäyttäjällä tarkoitetaan asiakasprojektin lopputuloksena syntynyttä Dynamics CRM -järjestelmää käyttävää asiakasyrityksen työntekijää. Virhetekstien näyttäminen tehdään heittämällä ohjelmakoodista *InvalidPluginExecutionException*-tyyppinen poikkeus [56]. Heitetyn poikkeuksen kuvausteksti näytetään sellaisenaan loppukäyttäjälle [56]. Ohjelmalistaus 3.1 selventää tätä. Kaikki muun tyyppiset poikkeukset aiheuttavat yleistetyn virheilmoituksen näyttämisen loppukäyttäjälle [56]. Ohjelmalistaus 3.2 näyttää tilanteen, jossa loppukäyttäjä näkee Dynamics CRM -järjestelmän automaattisesti muodostaman virheilmoituksen “On tapahtunut odottamaton virhe <tässä näytetään joitakin poikkeuksen tietoja>”, kun ohjelmakoodista heitetyn poikkeuksen tyyppi ei ole *InvalidPluginExecutionException*. Tällaiset virheilmoitukset eivät yleensä auta loppukäyttäjää, vaan ovat ennemminkin merkki puutteellisesta virhetilanteiden hallinnasta. Myös räätälöityjen työnkulun vaiheiden yhteydessä suorituksen keskeyttävä virhetilanne

ilmaistaan heittämällä ohjelmakoodista *InvalidPluginExecutionException*-tyyppinen poikkeus [40]. Liitännäisistä poiketen, tätä heitettyä poikkeusta ei kuitenkaan näytetä suoraan loppukäyttäjälle [40]. Johtuen tapahtumien käsittelyn liukuhihna-arkkitehtuurista, poikkeusten heittäminen on liitännäisten ja räätälöityjen työnkulun vaiheiden ainoa tapa pysäyttää käsittelyssä oleva transaktio [54]. Samalla tämä on ainoa tapa välittää omia virheilmoitustekstejä liitännäisistä tai räätälöidyistä työnkulun vaiheista loppukäyttäjälle.

```
public class ErrorMessagePlugin: IPlugin {
  public void Execute(...parametrit...) {
    throw new InvalidPluginExecutionException(
      "Liitännäinen on rekisteröity virheellisesti. " +
      "Ota yhteys järjestelmän toimittajaan.");
  }
}
```

Ohjelma 3.1 *Esimerkki loppukäyttäjälle näytettävän virheilmoituksen heittämisestä. Loppukäyttäjälle näytetään virheilmoitus "Liitännäinen on rekisteröity virheellisesti. Ota yhteys järjestelmän toimittajaan."*

```
public class UnexpectedErrorPlugin: IPlugin {
  public void Execute(...parametrit...) {
    throw new Exception(
      "Liitännäinen on rekisteröity virheellisesti. " +
      "Ota yhteys järjestelmän toimittajaan.");
  }
}
```

Ohjelma 3.2 *Esimerkki tilanteesta, jossa loppukäyttäjä näkisi virheilmoituksen "On tapahtunut odottamaton virhe liitännäisessä (Execute): UnexpectedErrorPlugin: System.Exception: Liitännäinen on rekisteröity virheellisesti. Ota yhteys järjestelmän toimittajaan."*

Dynamics CRM -kehitystyössä käytetään yleensä Dynamics CRM SDK -kirjastoja [74]. SDK (eng. *Software Development Kit*) on kokoelma ohjelmistokehitystyökaluja ohjelmiston toteuttamiseksi jollakin tietyllä kehitysalustalla [109, s. 934]. Tässä diplomityössä ei käsitellä muita kehitystyötapoja, koska diplomityössä rakennettavan toteutuksen kanssa on Dynamics CRM -alustan vaatimuksesta pakko käyttää Dynamics CRM SDK -kirjastoja. Eri Dynamics CRM -versioihin tehty ohjelmakoodi käännetään kyseisen Dynamics CRM -version SDK-kirjastoja vastaan. Näiden kirjastojen käyttämät .NET-versiot voivat vaihdella. Esimerkiksi Dynamics CRM

2013 SDK-kirjasto käyttää .NET-versiota 4.0 [75], kun taas Dynamics CRM 2015 ja Dynamics CRM 2016 SDK-kirjastot käyttävät .NET-versiota 4.5.2 [76; 77].

Dynamics CRM -järjestelmässä olevan tiedon käsittelyä tehdään lähes jokaisessa ohjelmoitavassa komponentissa. Dynamics SDK-kirjastot abstrahoivat järjestelmässä olevien entiteettien tietueet *Entity*-luokan instansseiksi [52]. Esimerkiksi aiemmin käsitellyn suorituskontekstin kautta saatavat tietueiden vedokset esitetään *Entity*-luokan instansseina [53; 59; 80]. Toinen esimerkki tämän luokan käytöstä on tietojen haku Dynamics CRM:n web service -rajapinnan kautta [63; 64]. Hakuehtoihin täsmänneet entiteetin tietueet palautetaan kutsujalle *Entity*-luokan instansseina. *Entity*-luokka toimii siis DTO-luokkana SDK-kirjastojen kanssa tehtävän ohjelmakoodin ja Dynamics CRM:n web service -rajapinnan välillä. DTO-luokka (eng. *Data Transfer Object*) kapseloi kahden järjestelmän välillä kuljetettavan tiedon [98; 11].

Entity-luokka sisältää yhden tietueen tietosisällön avain-arvopari -tyyppisessä kokoelmassa [52]. Kun entiteetin tietue esitetään *Entity*-luokan instanssina, ensimmäinen tarkistus siitä, vastaako instanssissa annettu tietosisältö järjestelmässä olevaa entiteetin metatietoa, tehdään ajonaikaisesti Dynamics CRM:n tapahtumankäsittelylinjastolla [86], eli tapahtuu dynaaminen sitominen [111, s. 116]. Ohjelmalistaus 3.3 havainnollistaa *Entity*-luokan instanssin alustamista avain-arvoparein. Tässä esimerkissä alustetaan *Contact*-entiteetin yhtä tietuetta kuvaava instanssi. Kun tässä diplomityössä puhutaan myöhemmin dynaamisesta ohjelmointitavasta, tarkoitetaan edellä kuvattua tapaa käyttää *Entity*-luokkaa tyypittämättömänä, dynaamisesti sidottavana DTO-luokkana.

```
Entity contact = new Entity("contact")
contact["firstname"] = "Teemu";
contact["lastname"] = "Teekkari";
```

Ohjelma 3.3 *Esimerkki dynaamisesti sidotun Entity-luokan käytöstä DTO-luokkana.*

Dynamics CRM:n SDK-kirjastot mahdollistavat myös käännösaikaisesti tyypitettävien DTO-luokkien käyttämisen [84]. Tämä käännösaikainen tyypitys käsitetään tässä diplomityössä staattiseksi sidonnaksi [111, s. 346]. Staattisesti tyypitettävät DTO-luokat periytetään *Entity*-luokasta [84]. Staattiset DTO-luokat sisältävä ohjelmakoodi luodaan Dynamics CRM SDK:n sisältämällä CrmSvcUtil.exe-työkalulla [45]. Tämä työkalu kirjoittaa ohjelmakoodia Dynamics CRM -järjestelmän tietomallia kuvaavan metatiedon perusteella. Luodussa ohjelmakoodissa jokainen järjestelmän entiteetti on yksi DTO-luokka [84]. Entiteettien kentät puolestaan on tyypitetty luo-

kan kentiksi käyttämällä tyyppiturvallisuuden takaavaa .NET-tietotyyppiä. Liitteessä A on katkelma CrmSvcUtil.exe-työkalulla luodun DTO-luokan ohjelmakoodista.

Ohjelmalistauksessa 3.4 on esimerkki staattisesti tyypitetyn DTO-luokan käytöstä. Tämä ohjelmalistaus on Dynamics CRM -tapahtumankäsittelylinjaston näkökulmasta täysin identtinen ohjelmalistauksessa 3.3 kuvatun dynaamisesti sidotun esimerkin kanssa. Suurin ero dynaamisen ja staattisen DTO-luokan käytön välillä on kääntäjän tekemä staattisten luokkien metadatan tyypityksen tarkistus [86]. Koska staattisesti tyypitetyt DTO-luokat periytetään dynaamisesti sidottavasta *Entity*-luokasta, Dynamics CRM -alusta tarkistaa kummassakin tapauksessa saamansa DTO-instanssin vastaavuuden järjestelmän metatietoon. Tämä on välttämätön tarkistus, sillä järjestelmän entiteettien metatiedot ovat voineet muuttua sen jälkeen kun DTO-luokat on luotu [84].

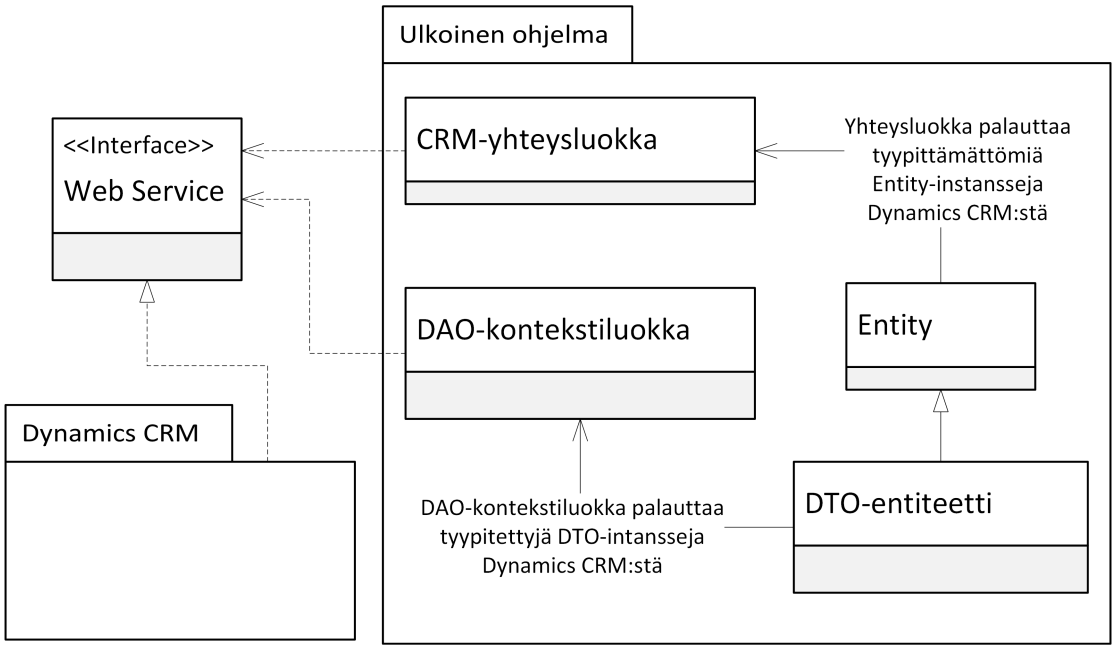
```
// Luokka Contact on periytetty luokasta Entity
Contact contact = new Contact() {
    FirstName = "Teemu",
    LastName = "Teekkari"
};
```

Ohjelma 3.4 *Esimerkki staattisesti sidotun, Entity-luokasta periytetyn Contact-luokan käytöstä DTO-luokkana. Katkelma Contact-luokan lähdekoodista on nähtävillä liitteessä A.*

Dynamics CRM:n entiteettien tietueita saa muun muassa luotua, muokattua ja kyseltyä web service -rajapinnan välityksellä [62; 63; 64; 65]. Kaikissa näissä rajapinnan operaatioissa käytetään dynaamisesti sidottua *Entity*-luokkaa käsiteltävän tietueen esittämiseksi. Dynamics CRM -kehitystyön ohjelmakoodissa on mahdollista tehdä samat operaatiot myös käyttämällä staattisesti sidottuja DTO-luokkia yhdessä DAO-kontekstiluokan kanssa [82; 83]. DAO-kontekstiluokka abstrahoi Dynamics CRM:n web service -rajapinnan operaatiot vahvasti tyypitetyiksi operaatioiksi, jotka huomioivat järjestelmän tietomallin metatiedot [42; 83]. Abstrahointi tarjotaan ohjelmistokehittäjälle sellaisen luokan avulla, jossa on vahvasti tyypitetyt LINQ-operaatiot tiedon luontiin, muokkaukseen ja hakemiseen. LINQ on .NET-kehyksen yleiskäyttöinen kyselykielirakenne tiedon kyselemiseksi [7]. LINQ tarjoaa yhtenäisen ohjelmointirajapinnan samojen operaatioiden tekemiseen taustalla olevasta tietolähteestä riippumatta. Termi on lyhennys englanninkielisestä nimestä *.NET Language-Integrated Query*.

DAO-kontekstiluokka sisältää yhden kentän jokaista Dynamics CRM:n tietomallin entiteettiä kohti [42]. Jokainen näistä kentistä mahdollistaa kyselyn tekemisen kyseisen entiteetin tietueisiin. Kyselyt ohjelmoidaan LINQ-kyselyiksi. Nämä muunnetaan

Dynamics CRM SDK-kirjaston toimesta automaattisesti vastaaviksi dynaamisesti tyypitetyiksi Dynamics CRM web service -rajapintaoperaatioiksi. Kuva 3.4 havainnollistaa sitä, miten Dynamics CRM:n web service -rajapinnasta voidaan kysellä tyypittämättömien *Entity*-instanssien lisäksi vahvasti tyypitettyjä DTO-instansseja käyttämällä DAO-kontekstiluokkaa. Kyselyiden rakentamisessa käytetään staattisesti tyypitettyjen DTO-luokkien kenttiä [82]. Ohjelmalistaus 3.5 antaa esimerkin DAO-kontekstiluokan käytöstä tietueiden kyselemiseen. Staattiset DTO-luokat luovalla CrmSvcUtil.exe-työkalulla saadaan luotua myös DAO-kontekstiluokan ohjelmakoodi [45]. Tämä kontekstiluokka periytyy aina *Microsoft.Xrm.Sdk.Client*-nimiavaruuden luokasta *OrganizationServiceContext* [45]. DAO-kontekstiluokan käytön on Accountor Enterprise Solutions Oy:n sisällä havaittu yleisesti nopeuttavan varsinkin yksinkertaisten Dynamics CRM -tietokantakyselyjen kirjoittamista. Lisäksi DAO-kontekstiluokan käytön on havaittu DTO-luokkien tavoin vähentävän ajonaikaisia dynaamisen tyypityksen virheitä. Kun tässä diplomityössä puhutaan myöhemmin staattisesta ohjelmointitavasta, tarkoitetaan käännösaikaisesti tyypitettyjen DTO-luokkien sekä DAO-kontekstiluokan käyttämistä ohjelmakoodia kirjoitettaessa.



Kuva 3.4 *Havainnollistus siitä, miten DAO-kontekstiluokka antaa mahdollisuuden käyttää Dynamics CRM:n tyypittämätöntä web-service rajapintaa vahvasti tyypitettynä. Kuvan esitysmuoto on UML:n luokkakaavio.*

```
// Oletetaan XrmServiceContext-luokan periytyvän
// luokasta OrganizationServiceContext.
var context = new XrmServiceContext();

List<Contact> results =
  context.ContactSet
         .Where(c => c.LastName == "Teekkari" &&
                     c.FirstName == "Teemu")
         .ToList();
```

Ohjelma 3.5 *Esimerkki tiedon hakemisesta DAO-kontekstiluokalla käyttämällä LINQ-kyselyä. Luokka XrmServiceContext on DAO-kontekstiluokka. Kyselyssä haetaan Dynamics CRM:stä kaikki hakuehtoihin osuvat Contact-entiteetin tietueet. Hakuehdoissa käytetään Contact-luokan kenttiä LastName ja FirstName.*

Dynamics CRM -alustan liitännäinen on sellainen luokka, joka toteuttaa rajapinnan *Microsoft.Xrm.Sdk.IPlugin* [89]. Dynamics CRM -alusta suorittaa kaikki järjestelmän työnkulut käyttäen Windows Workflow Foundation [87] -työnkulkukirjastoa [48]. Tämän kirjaston kautta määräytyy se, että jokainen räätälöity työnkulun vaihe periytetään luokasta *System.Activities.CodeActivity* [48]. Liitännäisten rajapinta ja räätälöityjen työnkulkujen kantaluokka kumpikin vaativat abstraktin *Execute*-metodin toteuttamista [89; 44]. Dynamics CRM:n tapahtumankäsittelylinjasto kutsuu toteutettua *Execute*-metodia käänteisen kontrollin periaatteen mukaisesti [43; 44; 66; 89]. Linjasto siis kutsuu metodia niiden liukuhihnan tapahtumien tapahtuessa, jotka ovat kiinnostavia rekisteröityjen liitännäisten ja työnkulkujen näkökulmasta.

Tapahtumankäsittelylinjaston suorituskontekstiin[3] liittyvät tiedot ovat tarjolla liitännäisille rajapinnan *Microsoft.Xrm.Sdk.IPluginExecutionContext* kautta [80]. Räätälöidyille työnkulun vaiheille vastaavat tiedot ovat tarjolla rajapinnan *IWorkflowContext* kautta [69]. Nämä kaksi rajapintaa ovat hyvin pitkälti yhtenevät [67; 69]. Yhtenevät osuudet on kapseloitu rajapinnaksi *IExecutionContext*, jonka sekä *IPluginExecutionContext* että *IWorkflowContext* toteuttavat. Nämä rajapinnat ovat ohjelmistokehittäjän saatavilla liitännäisen ja räätälöidyn työnkulun vaiheen *Execute()*-metodin saaman parametrin kautta [67; 73]. Ohjelmistokehittäjät hyödyntävät näiden rajapintojen sisältämiä tietoja jokaisessa liitännäisen tai räätälöidyn työnkulun vaiheen toteutuksessa.

[3]Suorituskonteksti on kuvattu tämän kohdan sivulla 16.

Dynamics CRM -alusta mahdollistaa erilaisten alustan tarjoamien vakiopalveluiden käytön sekä liitännäisistä että räätälöidyistä työnkulkujen vaiheista [80; 73]. Esimerkkejä näistä palveluista ovat muun muuassa *IOrganizationService* sekä *ITracingService* [80; 49]. Näistä *IOrganizationService* tarjoaa pääsyn Dynamics CRM:n web service -rajapinnan operaatioihin [85]. *ITracingService* puolestaan mahdollistaa liitännäisen suorituksen jäljittämisen kirjoittamalla lokiviestien tyyppisiä viestejä erityiseen rajapintaan [49]. Tähän palveluun kirjoitettuja viestejä ei kuitenkaan tallenneta mihinkään myöhempää analysointia varten, vaan tarkoituksena on yhden suorituksen jäljittäminen [49].

4. TOIMINNALLISUUS

Komponenttikehyksen suunnittelussa huomioitiin useita erilaisia asioita muun muuassa toiminnallisuuden, käytettävyyden ja teknisten rajoitteiden suhteen. Tässä luvussa kuvataan lähtökohdat, joista haluttuja toiminnallisuuksia lähdettiin kartoittamaan, kuvataan tehdyt oletukset ja muut suunnittelun rajoitukset sekä käydään läpi toiminnalliset vaatimukset.

4.1 Toiminnallisuuden suunnittelun lähtökohdat

Komponenttikehyksen toiminnalliset ja ei-toiminnalliset vaatimukset on koostettu diplomityön aikana useista eri näkökulmista ja useilta eri henkilöiltä. Vaatimuksiin ovat vaikuttaneet alkuperäiset ideat, kohdassa 2.2 kuvattujen haasteiden ratkaiseminen liiketoimintaa tukevalla tavalla sekä luvussa 7 kuvatun prosessin aikana tulleet uudet vaatimukset ja alkuperäisten ideoiden tarkentumiset. Komponenttikehyksen käyttäjiksi oletetaan Accountor Enterprise Solutions Oy:n tekniset arkkitehdit ja ohjelmistokehittäjät, jotka käyttävät komponenttikehystä asiakasprojektien toteutuksissa. Diplomityön lähtökohtaisena ydinajatuksena oli tarjota kantaluokat sekä liitännäisille että räätälöidyille työnkulun vaiheille. Tämän lisäksi ajatuksena oli tarjota valmiita toteutuksia auttamaan projekteissa säännöllisesti vastaan tulevissa tarpeissa, jotta tarve lähdekoodin kopioimiselle vähentyy. Liitännäisten ja räätälöityjen työnkulun vaiheiden kantaluokkien osalta diplomityön alkuperäiset ideat olivat seuraavat:

1. Keskitetty virhekäsittely.
2. Loppukäyttäjälle näytettävien virheilmoitusten harmonisointi.
3. Dynamics CRM:ään tallennettujen asetustietojen käsittely.
4. Tuki sekä staattisesti että dynaamisesti sidotuille ohjelmointitavoille.
5. Yksi rajapinta lokitukselle ja esimerkkitoteutukset suoralle tiedostoon lokittamiselle, Dynamics CRM:ään lokittamiselle sekä lokitus käyttäen NLog- [99] ja Enterprise Library [51] -lokituskirjastoja.

6. *PreImage*- ja *PostImage*-vedosten[1] nouto liitännäisen suorituskontekstista.

Vastaan tuleviksi haasteiksi tunnistettiin alkuvaiheessa neljä suurempaa teemaa. Toteutustekninen haaste tuli liiketoimintaympäristön siirtymisestä entistä enemmän pilvipalveluihin, joten katsottiin tarpeelliseksi tarjota myös pilviympäristössä toimiva komponenttikehys. Tästä kumpusi seuraava haaste, eli miten saadaan varmistettua se, ettei tehdä mitään sellaista, mikä ei toimi Dynamics CRM -pilviympäristössä ja ettei mikään tuleva muutos riko mitään jo tehtyä toiminnallisuutta. Lisäksi haastavaksi asiaksi tunnistettiin lopputuloksen tehokas jakeleminen yrityksen kaikkien Dynamics CRM -ohjelmistokehittäjien käyttöön. Viimeisimmäksi haasteeksi tunnistettiin tarve saada komponenttikehyksestä niin helppo käyttää, että kuka tahansa ohjelmistokehittäjä voisi lähteä kehittämään omien projektiensa kehitystyötä tätä komponenttikehystä käyttäen.

4.2 Oletukset järjestelmän arkkitehtuurista

Komponenttikehys olettaa, että kaikkia sen päälle toteuttavia laajennoksia ajetaan Dynamics CRM:n sisällä joko ennen tai jälkeen järjestelmän ydinoperaation tai laajennoksia kutsutaan ajettavaksi osana Dynamics CRM:n prosessi-nimellä käytettäviä vakiolaajennoksia. Toisin sanoen komponenttikehys olettaa kaikkien sen päälle toteutettavien laajennoksien olevan Dynamics CRM:n laajennusarkkitehtuurin [78] mukaisia liitännäisiä tai räätälöityjä työnkulun vaiheita.

Accountor Enterprise Solutions Oy:n toimittamat Dynamics CRM -järjestelmät rakennetaan auttamaan asiakasta heidän liiketoimintansa prosessien hoitamisessa. Tyypillisesti ne prosessien kohdat, joiden toteuttamiseen tarvitaan ohjelmoitua automaatiota, ovat vähintään kohtuullisessa määrin määriteltyjä ennen toteutuksen aloittamista. Komponenttikehyksen näkökulmasta voidaan siis olettaa, että järjestelmästä on jo tehty dekompositio liiketoiminnan näkökulmasta siinä vaiheessa, kun komponenttikehystä tullaan käyttämään. Lisäksi kohdassa 2.1 mainitun mukaisesti yksittäisten liitännäisten ja räätälöityjen työnkulun vaiheiden toteutukset ovat melko pieniä. Komponenttikehyksen toteutuksessa oletetaan sen päälle toteutettavien automaatioiden olevan pilkottu kohtuullisen pieniksi yksittäisiksi kokonaisuuksiksi.

Komponenttikehyksen kehitystyössä on tehty oletus siitä, ettei Dynamics CRM -järjestelmän arkkitehtuuri ja ohjelmointirajapinta juurikaan muutu siitä, mitä se on ollut Dynamics CRM 2011 -versioista eteenpäin. Arkkitehtuurin muutos ei kuitenkaan aiheuta suuria riskejä komponenttikehyksen päälle rakennetuille asiakasjärjes-

[1] Katso *PreImage* ja *PostImage* kohdasta 3.4 sivulta 18.

telmille, sillä tällöin komponenttikehyksestä voidaan toteuttaa muuttunutta arkkitehtuuria vastaava versio. Tällaisessa tilanteessa komponenttikehystä käyttävien asiakasjärjestelmien muutokset uuden arkkitehtuurin mukaiseksi saattavat jopa olla työmääräisesti pienempiä verrattuna tilanteeseen, jossa komponenttikehystä ei olisi käytetty.

Komponenttikehyksen käyttäjiksi oletetaan Accountor Enterprise Solutions Oy:n Dynamics CRM -projekteissa työskentelevät ohjelmistokehittäjät. Komponenttikehystä ylläpitäväksi sidosryhmäksi oletetaan sama ohjelmistokehittäjien joukko komponenttikehyksen omistajan ohjauksessa. Komponenttikehyksen omistajana toimii toistaiseksi tämän diplomityön kirjoittaja varmistamassa, etteivät liiketoiminnan tarpeet jää toteuttamatta komponenttikehyksen puutteiden vuoksi.

Komponenttikehyksen käyttöesimerkki Dynamics CRM -järjestelmän toteutuksessa on henkilön syntymäajan ja sukupuolen tallentaminen henkilön tietoihin automaattisesti, jos henkilölle annetaan oikean muotoinen henkilötunnus. Tällainen toiminto vaatii liitännäisen toteuttamisen. Liitännäisessä kutsuttaisiin henkilötunnuksen tarkistavaa ohjelmakoodia sekä syntymäajan ja sukupuolen päättelevää ohjelmakoodia. Komponenttikehyksen on tarkoitus auttaa ohjelmistokehittäjää niin, ettei hänen tarvitse toteuttaa liitännäistä varten käytännössä muuta kuin henkilötunnuksen tarkistuslogiikka sekä selvitetyn syntymäajan ja sukupuolen tallennus määriteltyihin henkilöä esittävän entiteetin kenttiin. Komponenttikehys tarjoaisi käytännössä kaiken tarvittavan infrastruktuuritason toteutuksen liitännäisen toteuttamiseksi. Ilman komponenttikehystä ohjelmistokehittäjän tulisi kirjoittaa toisteista koodia esimerkiksi lokituksen tai loppukäyttäjän asettaman kentän arvon hakemiseksi tapahtumankäsittelylinjaston sisältä.

4.3 Toiminnalliset vaatimukset

Komponenttikehyksen toiminnallisista vaatimuksista valtaosa kumpuaa kehitystyön alkuvaiheessa olleista ideoista sekä niiden tarkentumisesta lähinnä kevään 2014 aikana (ks. tarkemmin luvusta 7). Tärkeimpänä toiminnallisena vaatimuksena on yhteisten kantaluokkien tarjoaminen liitännäisten ja räätälöityjen työnkulkujen vaiheiden kehitykseen Dynamics CRM -alustalle. Näiden kantaluokkien tulee tarjota käyttäjälle mahdollisuus käyttää sekä staattisesti että dynaamisesti sidottuja ohjelmointitapoja.

Komponenttikehyksen tulee tarjota valmiit toteutukset yleisille liitännäisten ja räätälöityjen työnkulkujen vaiheiden toteutuksissa käytettäville toiminnoille. Esimerkki

yleisestä toiminnasta on liitännäisen suorituskontekstilta saatavan *PreImage*-vedoksen kenttien sisällön vertaaminen niihin arvoihin, joita suorituskontekstin käynnistäneessä operaatiossa on muutettu.

Virhetilanteiden hallintaan liittyy useita toiminnallisia vaatimuksia. Virhetilanteiden hallinnan lähtökohtana on kaikkien komponenttikehyksen alaisuudessa ajettavasta ohjelmakoodista kuplivien poikkeusten käsittely. Yhtään poikkeusta ei saa päästää kuplimaan komponenttikehyksen läpi ilman käsittelyä. Käsittelyn yhteydessä jokainen poikkeus tulee kirjoittaa lokiin. Lisäksi komponenttikehyksen tulee tarjota käyttäjälleen mahdollisuus kustomoida virheilmoitukset esimerkiksi asiakkaan tai loppukäyttäjän käyttämän käyttöliittymäkielen perusteella.

Useissa Accountor Enterprise Solutions Oy:n Dynamics CRM -projekteissa on käytetty tietokantaan luotua asetusentiteettiä järjestelmän automaatioiden käyttämien asetustietojen tallentamiseen. Komponenttikehyksen tulee tarjota mahdollisuus lukea asetusarvoja asetusentiteetissä olevista asetustietueista. Näiden tietueiden käsittelymahdollisuus tulee tarjota myös muualla kuin komponenttikehyksen suoraan kutsumasta liitännäisten ja räätälöityjen työnkulkujen vaiheiden ohjelmakoodista.

Komponenttikehyksen tulee tarjota käyttäjälle mahdollisuus kirjoittaa viestejä lokiin. Lokiin kirjoittamiseen tulee tarjota yhtenevä mekanismi myös muualta kuin komponenttikehyksen suoraan kutsumasta liitännäisten ja räätälöityjen työnkulkujen vaiheiden ohjelmakoodista, eli vastaavasti kuin myös asetustietueita pitää pystyä käsittelemään komponenttikehyksen ulkopuolella. Järjestelmän suorituksen ja varsinkin suorituksessa tapahtuneiden virhetoimintojen jäljittämisen vaatimuksesta komponenttikehyksen tulee kirjoittaa lokiin automaattisesti jokaisen liitännäisen tai räätälöidyn työnkulun vaiheen suorituksen aloitus ja lopetus. Komponenttikehyksessä tulee pystyä konfiguroimaan järjestelmäkohtaisesti se, minkä tasoiset lokiviestit päätyvät lokiin.

Käyttäjällä pitää olla komponenttikehyksen kautta helposti saatavilla tietyt liitännäisten ja räätälöityjen työnkulun vaiheiden suorituskontekstiin liittyvät palvelut ja tiedot. Tällaisia palveluita ovat esimerkiksi *IOrganizationService* ja *ITracingService*. Tarjottavia tietoja puolestaan ovat esimerkiksi *IPluginExecutionContext* ja *IWorkflowContext*.

Joskus on tarvetta jäljittää liian hitaasti toimivaa liitännäistä tai räätälöityä työnkulun vaihetta profiloimalla suoritusketjua ja suoritusten kestoa. Komponenttikehyksen tulee tarjota mahdollisuus profiloida liitännäisten ja räätälöityjen työnkulkujen suorituksia.

4.4 Ei-toiminnalliset vaatimukset

Komponenttikehykseen liittyy useita erilaisia ei-toiminnallisia vaatimuksia. Näistä monet liittyvät komponenttikehyksen konfiguroitavuuteen, virhetilanteiden hallintaan sekä yhteneviin käytäntöihin liitännäisten ja räätälöityjen työnkulun vaiheiden kesken. Lisäksi pitää huomioida taustalla vaikuttava tarve saada komponenttikehyksen käytöstä hyötyä myös liiketoiminnan näkökulmasta.

Liiketoiminnan näkökulmasta komponenttikehys ei saa ottaa kantaa siihen, millainen asiakaskohtainen toiminta sen päälle toteutetaan. Liiketoimintahyödyn näkökulmaa sivuaa myös vaatimus komponenttikehyksen käyttöönoton houkuttelevuudesta. Vaikka komponenttikehys olisi liiketoiminnan kannalta houkutteleva, sen käyttöön tulemiseen vaikuttaa lopulta se, kokevatko ohjelmistokehittäjät käyttöönoton riittävän houkuttelevaksi.

Komponenttikehyksen houkuttelevuuteen liittyy sen tarjoaman toiminnallisuuden lisäksi useita ei-toiminnallisia asioita. Monet näistä ovat käyttöönoton helppouteen ja muutosvastarinnan huomioimiseen liittyviä asioita.

Käyttöönoton helppouteen liittyvä vaatimus on saada komponenttikehyksen käyttöönotosta niin helppo, että jokaisen ohjelmistokehittäjän omien kirjastojen kopiointi uusiin projekteihin olisi työläämpi vaihtoehto. Tämän vaatimuksen painoarvo nousi suureksi, kun yrityksen sisällä keskusteltiin toimenpiteistä muutosvastarinnan vähentämiseksi. Tähän liittyy tarve hyvästä dokumentaatiosta. Kenen tahansa Accountor Enterprise Solutions Oy:n Dynamics CRM -ohjelmistokehittäjän pitää pystyä aloittamaan kehitystyö nojaten dokumentaatioon. Dokumentaatiossa pitää kuvata myös ne toiminnallisuudet, joissa komponenttikehys pystyy auttamaan ohjelmistokehittäjää hänen työssään.

Liiketoiminnan vaatimuksista voi katsoa kumpuavan myös vaatimus komponenttikehyksen toimimisesta sekä paikallisesti palvelimille asennetussa että pilvipohjaisessa Dynamics CRM -ympäristössä. Jo komponenttikehyksen ideointivaiheessa oli tunnistettu asiakkaiden tulevan tulevaisuudessa siirtyvän kohti pilvipohjaisia Dynamics CRM -ympäristöjä. Johtuen pilviympäristöjen tiukemmista rajoitteista komponenttikehyksessä ei saa olla mitään pilvessä toimimatonta toteutusta. Kun toteutus toimii pilviympäristössä, sen tiedetään toimivan myös paikallisissa ympäristöissä.

Komponenttikehyksen tulee käsitellä tapahtuneet poikkeukset keskitetysti. Tämä vaatimus tulee ylläpidollisesta näkökulmasta. Muussa tapauksessa virhekäsittely hajaantuu ympäri sovellusta niin, että ylläpito hankaloituu huomattavan paljon. Poikkeuksien käsittelyssä pitää huomoida se, ettei loppukäyttäjällä ole aina mahdolli-

suutta toistaa sovelluksesta saamaansa virhettä. Tämän takia tulee pyrkiä näyttämään loppukäyttäjälle sellainen virheilmoitus, jossa on jo yksinään mahdollisuuksien mukaan riittävästi tietoa virheen jäljittämisen aloittamiseksi. Loppukäyttäjän nähtäväksi päätyvät poikkeukset tulee harmonisoida niin, että loppukäyttäjä näkee yhtenäisesti muotoillut virheilmoitukset. Poikkeus pitää päästää sellaisenaan läpi, jos sen tyyppi on *InvalidPluginExecutionException*. Tämä Dynamics CRM:n oma poikkeustyyppi tulkitaan sellaiseksi, joka sisältää virheilmoituksen muotoilun juuri sellaiseksi kuin sen halutaan näkyvän.

Myös komponenttikehyksen tarjoamaan lokitukseen liittyy useita ei-toiminnallisia vaatimuksia. Syntyvän lokitiedoston muodon tulee olla ohjelmallista jäsentämistä tukeva. Jokaiseen lokiin kirjoitettavaan riviin on liitettävä mukaan myös liitännäisen sen hetkisen suorituskontekstin rekursiosyvyys. Rekursiosyvyys on tietyissä tilanteissa oleellinen tieto virheellisten tapahtumaketjujen tai suorituskykyongelmien selvittämisen kannalta. Lisäksi jokaisesta lokiviestistä pitää saada korrelaatiotunnuksen avulla tieto siitä mihin suorituskontekstiin kyseinen rivi liittyy. Korrelaatiotunnuksen avulla saadaan tarvittaessa selvitettyä kaikki samaan suorituskontekstiin liittyvät lokirivit.

Joitakin ei-toiminnallisia vaatimuksia tulee myös ylläpidettävyyden ja toteutuksen modulaarisuuden suunnalta. Komponenttikehyksen toteutuksessa tulee huomioida se, ettei mikään komponenttikehyksen päälle rakennettu asiakkaan järjestelmän räätälöinti ala toimia virheellisesti jonkin komponenttikehyksen uuden ominaisuuden tai muutoksen takia. Komponenttikehyksen käyttäjän näkökulmasta kehitystyön tulee olla samanlaista sekä liitännäisiä että räätälöityjä työnkulun vaiheita kehitettäessä. Lisäksi komponenttikehyksen toteutuksessa käytettyjen staattisesti sidottujen DTO-luokkien pitää voida sijaita myös muussa kuin siinä kirjastossa, josta liitännäistä tai räätälöityä työnkulun vaihetta ollaan suorittamissa. Kirjasto tarkoittaa tässä käytännössä .NET-kääntäjän tuottamaa dll-tiedostoa.

5. KOMPONENTTIKEHYKSEN RAKENNE JA ERIKOISTAMINEN

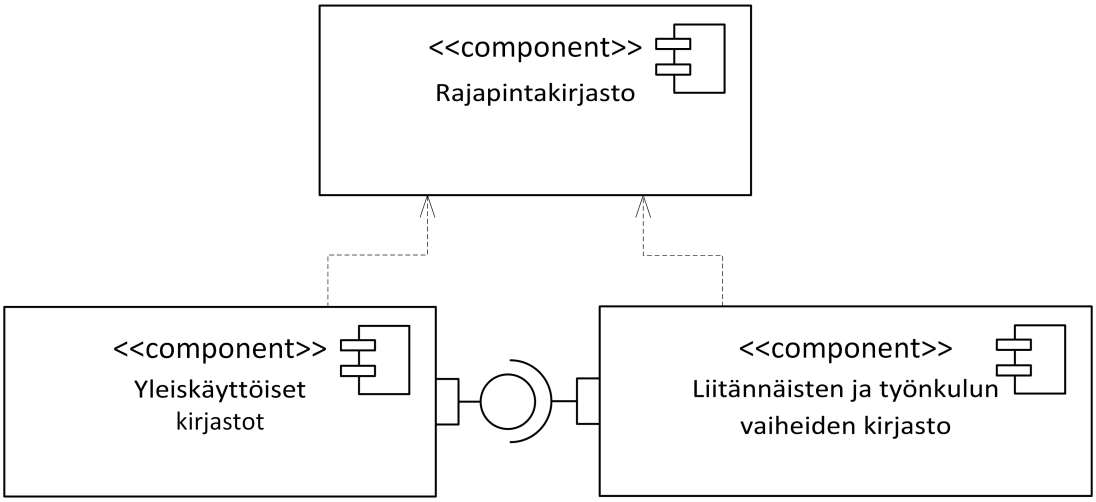
Tässä luvussa kuvataan tarkemmin komponenttikehyksen rakennetta. Luvussa kuvataan, miten komponenttikehys jakautuu kirjastoihin ja luokkiin. Lisäksi kuvataan, miten komponenttikehys erikoistetaan asiakastarpeeseen ja mitä komponenttikehys tarjoaa Dynamics CRM -alustan tapahtumalinjaston tapahtuman käsittelyyn liitännäisiä ja räärälöityjä työnkulun aktiviteetteja varten.

5.1 Kirjastorakenne

Komponenttikehyksen kirjastorakenne on ositettu toiminnallisuuden ja tiedon abstrahoinnin näkökulmista. Komponenttikehys jakautuu karkeasti rajapintakirjastoon, liitännäisten ja työnkulun vaiheiden toteutuskirjastoon sekä yleiskäyttöisiin kirjastoihin. Näiden välisiä suhteita havainnollistaa kuva 5.1. Kuvasta nähdään, miten rajapintakirjasto on tehty purkamaan konkreettisten toteutusten välisiä riippuvuuksia. Kuvasta nähdään myös, miten liitännäisten ja työnkulun vaiheiden kirjasto vaatii tiettyjen rajapintojen toteutukset. Tämä rajapintavaatimus on komponenttikehyksen konfiguroinnin erikoistamisrajapinta. Kuvassa olevat yleiskäyttöiset kirjastot tarjoavat komponenttikehyksen erikoistamisrajapinnan vaatimat oletustoteutukset.

Kuvasta 5.2 puolestaan nähdään ylätason kuvauksena, miten komponenttikehystä on tarkoitus käyttää asiakaskohtaisessa järjestelmässä. Asiakaskohtainen sovitus sisältää komponenttikehyksen konfiguroinnin erikoistuksen tiettyyn asiakasympäristöön. Tämä sovitus voi sisältää sellaisia erikoistamistoteutuksia, jotka poikkeavat komponenttikehyksen sisältämistä erikoistamisen oletustoteutuksista. Komponenttikehykselle voidaan tarjota samanaikaisesti erikoistamisen oletustoteutuksia ja asiakaskohtaisia erikoistamistoteutuksia. Asiakaskohtainen logiikka toteutetaan hyödyntämällä komponenttikehyksen asiakaskohtaista sovitusta.

Komponenttikehyksen toteutunut kirjastorakenne on esitetty kuvassa 5.3. Vertailemalla kuvia 5.3 ja 5.1, havaitaan toteutettu ositus kirjastoihin. Komponenttikehyksen kirjastoista käytetään tässä diplomityössä nimiä *Interfaces*, *ExtensionBase*,



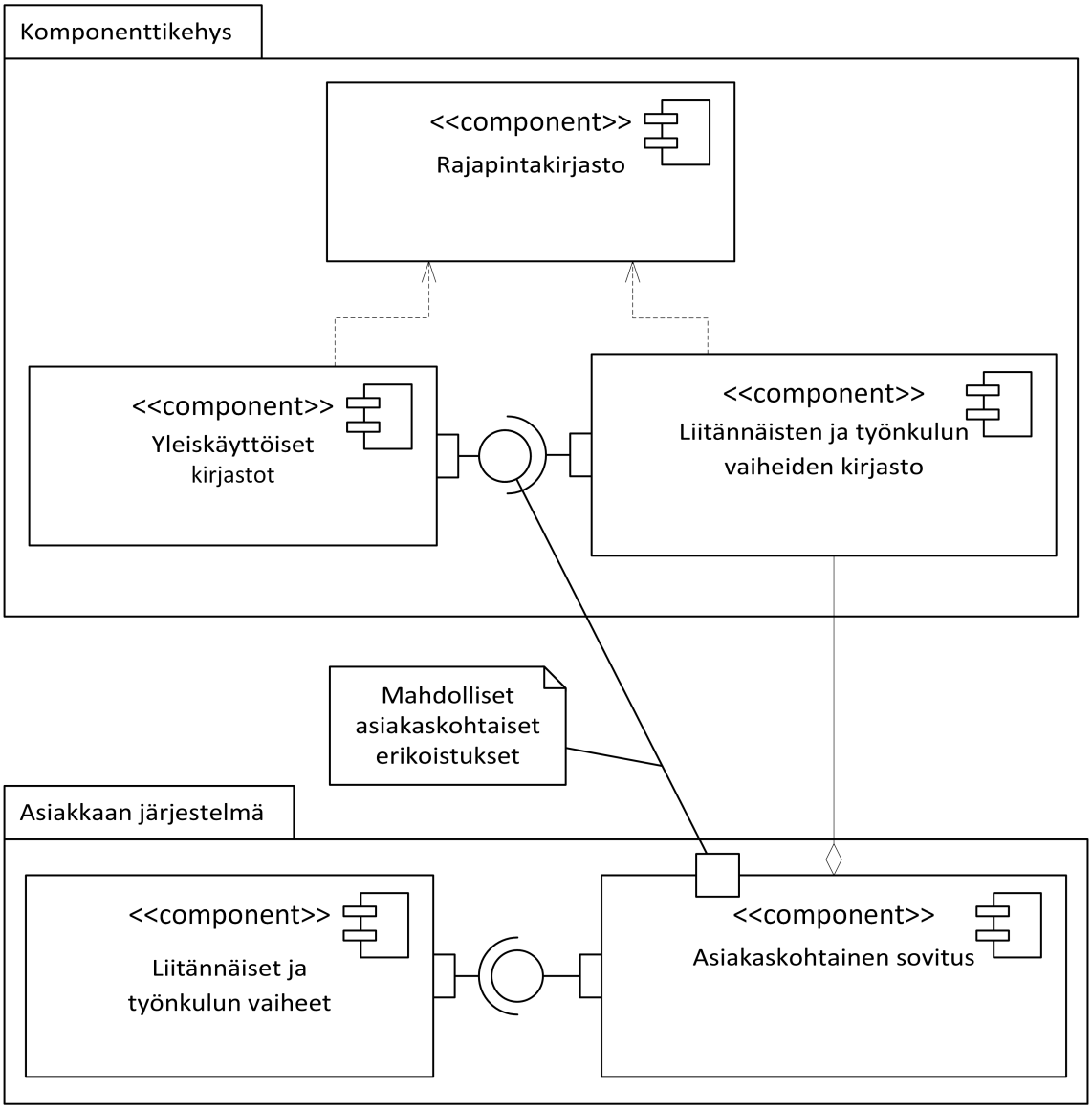
Kuva 5.1 *Komponenttikehyksen yhteydessä muodostuneiden kirjastojen karkea jaottelu sekä kirjastojen väliset suhteet UML:n komponenttikaaviona.*

Common ja *Logger.NLog*. Näiden nimien alusta on poistettu toistuva osuus, joka sisältää esimerkiksi yrityksen nimen. Seuraavaksi toteutunutta kirjastorakennetta esitellään tarkemmmin.

Komponenttikehyksen ytimen muodostavat rajapintakirjasto *Interfaces* ja liitännäisten ja työnkulun vaiheiden toteutuskirjasto *ExtensionBase*. *ExtensionBase*-kirjastossa on komponenttikehyksen käytön kannalta konkreettisimmaksi miellettävä toteutus. Tämä tarkoittaa kantaluokkia, joista asiakaskohtaiset liitännäiset ja työnkulun vaiheet periytetään. Tämän kirjaston yksityisessä toteutuksessa on paljon sellaista, joka mahdollistaa kohdissa 5.3 ja 5.4 kuvattujen asioiden toteutumisen. Komponenttikehyksessä on kantaluokat sekä liitännäisille että räätälöidyille työnkulun vaiheille. Näistä molemmista on tarjolla dynaamisen ja staattisen ohjelmointitavan[1] versiot, eli kantaluokkia on yhteensä neljä. Näiden kantaluokkien keskinäiset erot on kuvattu liitteessä B. Nämä erot eivät ole merkittäviä tässä kohdassa käsiteltävän asian kannalta.

Kirjastossa *Interfaces* on komponenttikehyksen käyttöä tukevien ja riippuvuuksia purkavien rajapintojen esittelyt. Kirjasto sisältää sellaiset rajapinnat, jotka ovat olleet tarpeen halutun toiminnalliset ja ei-toiminnalliset vaatimukset täyttävän komponenttikehyksen toteuttamiseksi. Tämän kirjaston tärkein tehtävä on mahdollistaa asiakaskohtainen erikoistaminen sekä komponenttikehyksen sisäisen toteutuksen piilottaminen. Komponenttikehyksen koko ulkoinen rajapinta määritellään käyttämällä tämän kirjaston sisältämiä rajapintakuvauksia. Tämän kirjaston rajapinnoilla on tärkeä tehtävä myös komponenttikehyksen dokumentoinnissa. Jokainen tämän kir-

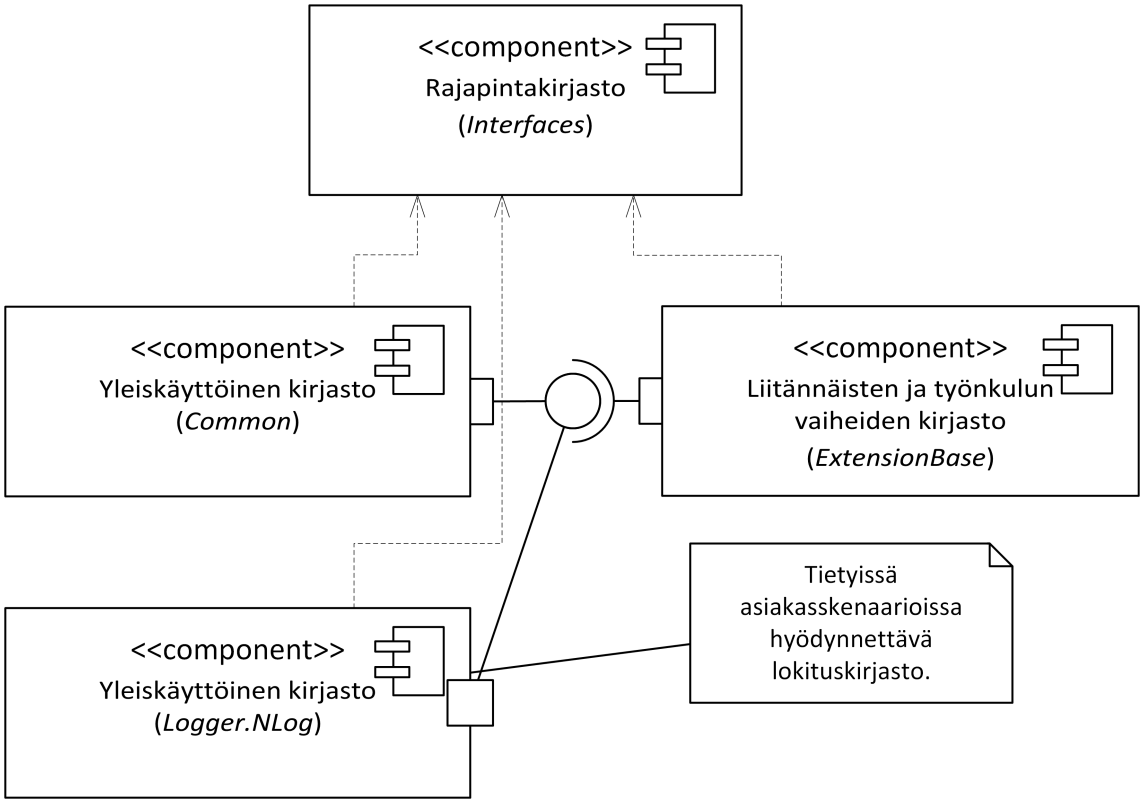
[1] Katso dynaaminen ja staattinen ohjelmointitapa kohdan 3.4 sivuilta 22 ja 24.



Kuva 5.2 *Komponenttikehyksen kirjastojen käyttö asiakaskohtaisessa toteutuksessa UML:n komponenttikaaviona. Kuvassa esitetään tilanne, jossa erikoistaminen on tehty osittain asiakaskohtaisesti ja osittain käytetään yleiskäyttöistä oletustoteusta.*

jaston tarjoama rajapinta ja rajapintametodi on dokumentoitu ohjelmistokehittäjän työtä tukevien IntelliSense-kommenttien avulla.

Komponenttikehyksen kantaluokkien ja rajapintojen lisäksi diplomityössä rakentui kaksi kirjastoa komponenttikehyksen erikoistamiseen. Nämä kirjastot ovat nimiltään *Common* ja *Logger.NLog*. Nämä kirjastot ovat luonteeltaan komponenttikehyksen kantaluokkia tukevia kirjastoja. *Common*-kirjasto sisältää muun muuassa *ExtensionBase*-kirjaston erikoistamiseen käytetyt oletustoteutukset. Kirjasto *Logger.NLog* puolestaan tarjoaa lokin kirjoittamiseen komponenttikehyksen rajapintojen mukaisen sovittimen. Näitä kirjastoja voidaan hyödyntää myös muissa Dynamics CRM



Kuva 5.3 *Komponenttikehyksen toteutunut kirjastorakenne sekä kirjastojen väliset suhteet UML:n komponenttikaaviona. Komponenttien nimien perässä suluissa ovat toteutettujen kirjastojen todelliset nimet.*

-kehitystöissä kuin liitännäisten ja räätälöityjen työnkulun vaiheiden toteuttamisessa. Nämä kirjastot eivät siis sisällä mitään sellaista toteutusta, josta periytetään asiakaskohtaisia liitännäisiä tai räätälöityjä työnkulun vaiheita.

Kirjasto *Common* sisältää oletustoteutukset asetustietueiden ja virheiden käsittelyyn. Virheillä tarkoitetaan tässä yhteydessä ajonaikaisia poikkeuksia. Kirjaston sisältämiä oletustoteutuksia voidaan käyttää komponenttikehyksen erikoistamisen oletustoteutuksina. Nämä oletustoteutukset ovat komponenttikehyksen vaatimia minimitoteutuksia, mutta ne ovat myös esimerkkejä asiakaskohtaiselle erikoistamiselle. Näiden oletustoteutusten erikoistaminen on suositeltavaa. Asetustietueiden osalta *Common*-kirjaston sisältämä oletustoteutus ei vaadi asiakasprojekteissa ohjelmakoodilla tehtävää erikoistusta, vaan yksinkertaisen konfiguroinnin. Oletustoteutus käsittelee asetustietueita avain-arvopareina. Lisäksi *Common*-kirjasto sisältää muitakin kehitystyössä hyödynnettäviä ratkaisuja. Nämä muut ratkaisut eivät ole oleellisia komponenttikehyksen erikoistamisen kannalta.

Kirjasto *Logger.NLog* sisältää toteutuksen lokiviestien kirjoittamiselle NLog [99] -lokituskirjastolla. Kirjasto toimii sovittimena NLog-kirjaston ja komponenttikehyksen

yleisen lokitusrajapinnan välillä. Komponenttikehystä erikoistettaessa lokitukseen voidaan valita tämä valmis esimerkkikirjasto. Tämä kirjasto toimii myös esimerkkinä toteutettaessa muita vastaavia sovittimia eri lokituskirjastoihin.

5.2 Asiakaskohtaisen konfiguroinnin erikoistamisrajapinta

Tässä kohdassa käsitellään komponenttikehyksen konfigurointi asiakasprojektissa käytettäväksi. Komponenttikehyksen konfiguraation erikoistamisen tekee yleensä asiakasprojektin tekninen arkkitehti. Tekninen arkkitehti erikoistaa asiakaskohtaiset abstraktit kantaluokat liitännäisille ja räätälöidyille työnkulun vaiheille. Asiakasprojektin kantaluokkien erikoistaminen tehdään periyttämällä ne komponenttikehyksen kantaluokista.

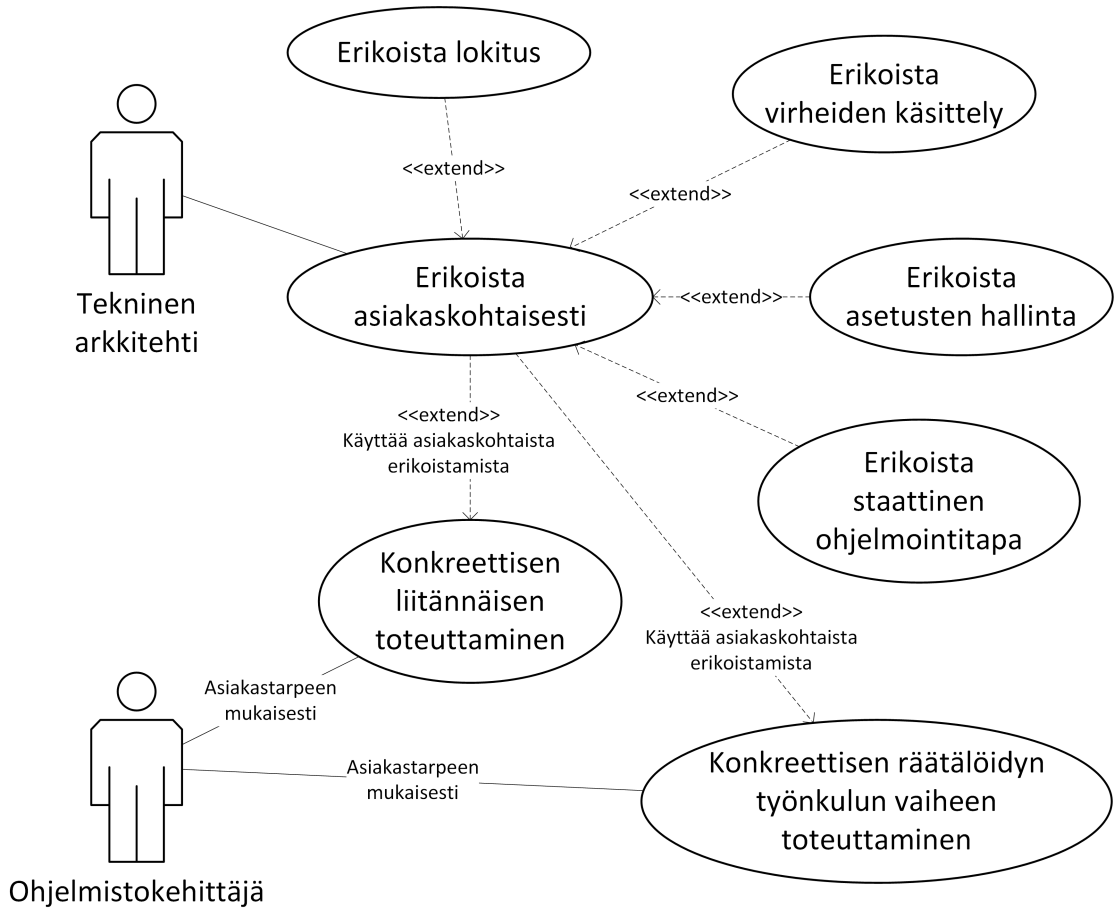
Komponenttikehys tarjoaa valmiiksi erikoistetut esimerkit asiakaskohtaisista kantaluokista. Myös asiakaskohtaiseksi erikoistetut kantaluokat on tarkoitettu abstrakteiksi luokiksi. Komponenttikehyksen erikoistamisrajapinnat ovat samanlaiset sekä liitännäisten että räätälöityjen työnkulkujen vaiheiden kantaluokkien osalta. Sama asiakaskohtainen erikoistamistoteutus käy näistä molempiin. Komponenttikehyksen kantaluokkien rakentajille annetaan parametreina asiakasprojektin vaatimusten mukaiset erikoistamisrajapintojen toteutukset. Toisin sanoen, erikoistamiseen käytetään riippuvuuksien injektointia rakentajan kautta. Komponenttikehyksen kantaluokkien rakentajista on tarjolla kuormitetut versiot. Kuormitetuilla rakentajilla voidaan valita asiakasprojektin käyttöön vain tietyt komponenttikehyksen ominaisuudet.

Erikoistamisen jälkeen ohjelmistokehittäjät toteuttavat asiakaskohtaisen sovelluslogiikan hyödyntämällä asiakaskohtaisesti erikoistettuja kantaluokkia. Ohjelmistokehittäjä tekee tämän periyttämällä sovelluslogiikkaa varten konkreettiset luokat asiakasprojektiin konfiguroiduista liitännäisen ja räätälöidyn työnkulun vaiheen kantaluokista. Tämän jälkeen hänen pitää toteuttaa haluttu sovelluslogiikka. Komponenttikehyksen logiikan erikoistaminen asiakastarpeeseen käsitellään kohdassa 5.3.

Komponenttikehyksessä on nykyisellään neljä kuvan 5.4 mukaista erilaista erikoistamistapausta. Nämä ovat lokituksen, virheiden käsittelyn, asetusten hallinnan sekä käytettävän staattisen ohjelmointitavan[2] DAO-kontekstiluokan erikoistaminen. Näiden käyttötapausten erikoistaminen tehdään erillisten rajapintojen kautta. Komponenttikehys vaatii virheenkäsittelyn ja lokituksen erikoistamisen. Nämä on koettu niin tärkeiksi, ettei komponenttikehyksen käyttöä ilman näitä haluta sallia. Muilta osin erikoistaminen tehdään, mikäli se on tarpeen kyseisessä asiakasprojektissa.

[2]Katso staattinen ohjelmointitapa sekä DAO-kontekstiluokka kohdan 3.4 sivuilta 23 ja 24.

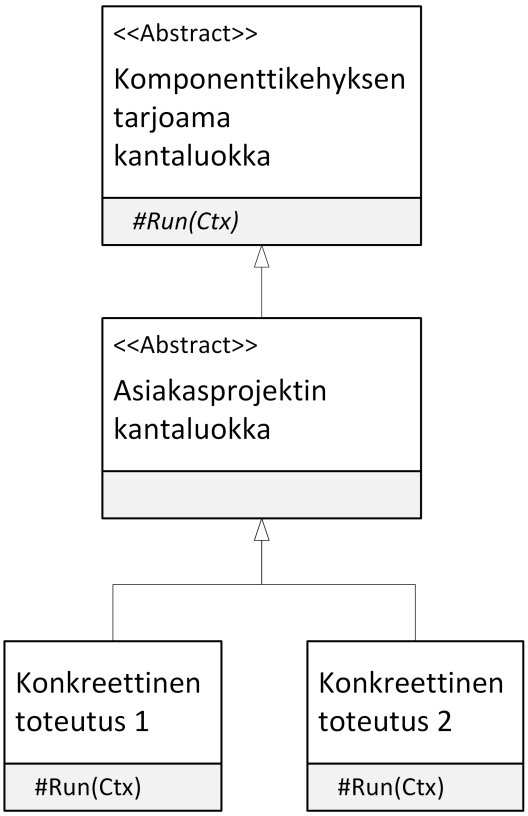
Komponenttikehyksen kantaluokissa ei ole rakentajia, joilla erikoistamisen voisi tehdä ilman virheenkäsittelyä ja lokitusta. Komponenttikehyksen erikoistamisen käyttötapauksia vastaavat oletustoteutukset ja niiden käyttö on esitetty yksityiskohtaisemmin liitteessä C.



Kuva 5.4 *Komponenttikehyksen erikoistamisen käyttötapaukset UML:n käyttötapauskaaviona. Teknisen arkkitehdin tekemän komponenttikehyksen erikoistamisen jälkeen ohjelmistokehittäjä käyttää erikoistamisen tulosta konkreettisiin toteutuksiin.*

Komponenttikehyksen rakenne ja erikoistamisrajapinta mahdollistavat asiakasprojektin käyttämän erikoistuksen muuttamisen ilman yhdenkään konkreettisen luokan muuttamista. Tämä on mahdollista komponenttikehyksen kantaluokkien, asiakaskohtaisten kantaluokkien ja konkreettisten sovelluslogiikkaluokkien keskinäisen periytymishierarkian sekä erikoistamisessa käytettyjen rajapinta-abstraktioiden ansiosta. Tämä periytymishierarkia voidaan yleistettynä esittää kuvan 5.5 mukaisesti. Asiakaskohtainen kantaluokka on hierarkiassa omana tasonaan. Näin saadaan mahdollisuus muuttaa asiakasprojektin kantaluokkaa ilman vaikutuksia mihinkään muualle. Muutokset voivat tarkoittaa esimerkiksi erilaista virheenkäsittelymekanismia testi- ja tuotantoympäristöjen kesken tai suorituksen profilointituen käyttöönottoa, jotta testiympäristössä tehtävä suorituskykytestausta helpottuu. Asiakaskoh-

taisen erikoistuksen muuttamisen jälkeen konkreettiset toteutukset tulee kääntää ja asentaa uudelleen asiakkaan ympäristöön.



Kuva 5.5 *Ylätasolla kuvattu periytymishierarkia tilanteesta, jossa asiakasprojektiin on toteutettu kaksi konkreettista asiakaskohtaista sovelluslogiikan toteutusta käyttämällä hyväksi komponenttikehyksen tarjoamaa kantaluokkaa. Esitysmuotona on UML:n luokkakaavio.*

Komponenttikehyksen erikoistamista ei käytännössä ole järkevää tehdä ajonaikaisesti. Ajoaikaista erikoistamista olisi esimerkiksi tarvittavien konfiguraatiotietojen lukeminen tiedostosta tai tietokannasta. Näistä molemmat tavat ovat hitaita verrattuna käännösaikaiseen erikoistamiseen. Käytännössä konfiguraatiotiedot pitäisi tällöin lukea jokaisen konkreettisen toteutuksen ensimmäisellä suorituskerralla. Tämä lisäisi vältettävissä olevaa loppukäyttäjän kokemaa viivettä, koska Dynamics CRM instantioi konkreettiset luokat reaktiona johonkin loppukäyttäjän toimenpiteeseen. Nämä tavat myös lisäävät mahdollisuuksia ajonaikaisille erikoistamisvirheille esimerkiksi tietoliikennevirheiden tai konfiguraatiossa olevan kirjoitusvirheen takia. Teknisesti tiedostosta lukemisen estää Dynamics CRM:n pilviympäristö, joka ei salli levyjärjestelmän käyttöä. Myös Dynamics CRM -järjestelmän sisäisesti käyttämä olioallas-suunnittelumalli tekisi erikoistamistiedon ajonaikaisesta lukemisesta haastavaa. Dynamics CRM säilöö ja uudelleenkäyttää luomiaan instansseja tämän suunnittelumallin mukaisesti määräämättömän ajan. Tämä tekee konfiguraa-

tion muuttamisesta hankalaa, koska ei tiedettäisi, milloin uusi konfiguraatio astuu voimaan.

5.3 Asiakaskohtaisen logiikan erikoistamisrajapinta

Asiakaskohtaisella logiikalla tarkoitetaan sellaista ohjelmoitua toteutusta, joka kohdan 4.2 mukaisesti auttaa Dynamics CRM -järjestelmää käyttävän asiakkaan liiketoimintaa. Komponenttikehys kutsuu päällensä toteutettua asiakaskohtaista logiikkaa käänteisen kontrollin periaatteen mukaisesti. Asiakaskohtainen logiikka toteutetaan kantaluokassa esiteltyyn abstraktiin metodiin. Komponenttikehys kutsuu tätä metodia tietyssä suorituksen kohdassa. Ideana tämä on melko lähellä sitä, miten Dynamics CRM vaatii liitännäisiä ja räätälöityjä työnkulun vaiheita toteuttamaan *Execute()*-metodin ja sen jälkeen kutsuu tätä metodia oikeassa tapahtumankäsittelylinjaston vaiheessa[3].

Komponenttikehyksen asiakaskohtainen logiikka toteutetaan metodissa *Run()*. Konkreettinen toteutus ohjelmoidaan komponenttikehyksestä periytettyihin lapsiluokkiin. Komponenttikehys kutsuu tätä metodia Dynamic CRM:n kutsuman *Execute()*-metodin sisältä. Tätä selventää ohjelmalistaus 5.1, jossa on kuvattu, miten *Execute()*-metodi on toteutettu komponenttikehyksen kantaluokassa. Käytännössä *Run()*-metodi on kohdassa 3.3 selitetty ohjelmistokehyksen sovelluskohtaisen ohjelmakoodin laajennuskohta.

Komponenttikehyksen kantaluokissa metodi *Run()* on esitelty abstraktina. Tämä rakenne vaatii jokaista komponenttikehyksen käyttäjää toteuttamaan sovelluskohtaisen laajennoksen samassa paikassa. Tämän takia sovelluslogiikalla ei ole mitään vaikutusta komponenttikehyksen toimintaan. Metodin *Run()* sijaintia toteutuksen periytymishierarkiassa selventää kuva 5.5 edellisen kohdan sivulla 39.

Asiakaskohtaisen logiikan laajennuskohtana toimiva metodi *Run()* on ympäröity sekä profiloinnin että poikkeusten käsittelyn mahdollistavalla toteutuksella. Komponenttikehyksen käyttäjä ei näe näitä toteutuksia. Poikkeusten käsittelyllä taataan kaikkien tapahtuvien poikkeusten kirjaaminen lokiin ilman, että käyttäjän pitää itse kirjoittaa näitä lokiin. Poikkeusten käsittelyssä käytetään kohdassa 5.2 kuvattua asiakaskohtaisesti erikoistettua virheenkäsittelyä. Jokainen asiakaskohtaisen logiikan heittämä poikkeus otetaan kiinni ja annetaan virheenkäsittelijän käsiteltäväksi. Poikkeuksen kirjaaminen lokiin ei riipu tästä virheenkäsittelijästä.

[3]Katso *Execute()*-metodi ja sen kutsuminen kohdan 3.4 sivulta 25.

```
// Komponenttikehyksen kantaluokan toteutuksesta on poistettu
// esimerkin kannalta tarpeettomat kohdat.
public abstract class PluginBase {

  // Dynamics CRM kutsuu tätä automaattisesti.
  public void Execute(IServiceProvider serviceProvider) {

    // Tässä kohdassa komponenttikehys toteuttaa oman
       alustuksensa.

    // Suorita sovelluskohtainen laajennoskoodi.
    Run(...parametrit...);

    // Tässä kohdassa komponenttikehys toteuttaa
    // sovelluskohtaisen laajennoksen suorituksen
    // jälkeisiä osiaan.
  }

  // Tämä metodi on sovelluskohtainen laajennoskohta.
  protected abstract void Run(...parametrit...);
}
```

Ohjelma 5.1 *Havainnollistus siitä, miten komponenttikehys kutsuu sovelluskohtaisen laajennuskohdan toteutusta. Tästä ohjelmalistauksesta on havainnollisuuden vuoksi poistettu toteutusyksityiskohtia.*

Komponenttikehyksen käyttäjä voi hyödyntää profilointia tarvitessaan erilaisia asiakaskohtaisen logiikan suoritukseen liittyviä telemetriikkatietoja. Näitä tietoja hyödynnetään yleensä suorituskykyongelmissa pullonkaulojen selvittämiseen.

5.4 Liitännäisille ja räätälöityjen työnkulun vaiheille tarjottava komponenttikehyskonteksti

Komponenttikehyksen täytyy välittää ajonaikaista tietoa siitä periytettyjen luokkien instansseille. Tiedot välitetään parametrina periytettyjen luokkien *Run()*-metodille. Tämä parametri välittää tietoina Dynamics CRM -alustan tapahtumankäsittelylinjaston suorituskontekstin, tämän suorituskontekstin kanssa operointia helpottavia metodeja sekä muita liitännäisten ja räätälöityjen työnkulun vaiheiden toteutuksia helpottavia metodeja. *Run()*-metodille välitettävästä parametrista käytetään jatkos-

sa termiä komponenttikehyskonteksti, jotta sitä ei sekoiteta Dynamics CRM -alustan yhteydessä käytettyyn suorituskonteksti-termiin.

Komponenttikehyskontekstista on neljä eri rajapintavariaatiota. Tämä jako roolirajapintoihin noudattaa komponenttikehyksen kantaluokkarakennetta. Komponenttikehyksen kantaluokkarakenne on nähtävissä liitteestä B. Jokaiselle kantaluokalle on erityyppinen komponenttikehyskonteksti. Kaikki komponenttikehyskontekstit periytetään samasta, yhteiset toiminnot sisältävästä roolirajapinnasta. Kaikkien variaatioiden kautta tarjotaan mahdollisuus käyttää Dynamics CRM -alustan suorituskontekstiin liittyviä Dynamics CRM:n vakiorajapintoja.

Ohjelmistokehittäjä pystyy käyttämään dynaamista ohjelmointitapaa[4] kaikkien komponenttikehyskontekstivariaatioiden kanssa. Komponenttikehyskonteksteista on tarjolla myös staattisen ohjelmointitavan variantit. Näitä käytetään vastaavien kantaluokkavarianttien kanssa. Esimerkiksi siis liitännäisille tarkoitettu staattisen ohjelmointitavan kantaluokka käyttää staattisen ohjelmointitavan komponenttikehyskontekstia. Staattisen ohjelmointitavan komponenttikehyskontekstit tarjoavat ohjelmistokehittäjälle asiakaskohtaisesti erikoistetun DAO-kontekstiluokan instanssin. Ohjelmistokehittäjän kannalta tämä DAO-kontekstiluokka on yhtä helposti saatavilla kuin esimerkiksi Dynamics CRM -alustan tiedon haun vakiorajapinta *IOrganizationService*. Näistä molemmat löytyvät komponenttikehyskontekstin luokasta omina kenttinään.

Dynamics CRM -alustan vakiorajapintojen lisäksi komponenttikehyskontekstin kautta on saatavilla asiakasprojektin mukaiseksi erikoistettu asetustietueiden haku ja lokiin kirjoitus. Kaikki komponenttikehyskontekstin variaatiot mahdollistavat asiakasprojektissa käytössä olevien asetustietueiden hakemisen. Komponenttikehyskonteksti tarjoaa mahdollisuuden kirjoittaa lokiin sanomia, jotka on uniikisti yksilöity liittymään kyseiseen tapahtumankäsittelylinjaston tapahtumaan. Yksilöinti tapahtuu jokaisen lokiviestin kirjoituksen yhteydessä. Jokainen lokiviesti liitetään automaattisesti myös Dynamics CRM:n omaan *ITracingService*-viestivirtaan.

Liitännäisille tarjottavassa komponenttikehyskontekstissa on metodit *PreImage*- ja *PostImage*-vedoksien[5] noutamiseksi Dynamics CRM -alustan suorituskontekstista. Näiden vedosten pitää olla nimetty sen liitännäisen luokan mukaan, josta vedosta noudetaan. Vedoksen nimessä pitää olla jälkiliitteenä “Pre” tai “Post” riippuen siitä, kumpaa vedosta käytetään. Esimerkiksi siis liitännäiselle *QuaranteeMarginPlugin* konfiguroitavan *PreImage*-vedoksen nimi olisi “QuaranteeMarginPluginPre”. Syy tä-

[4]Katso dynaaminen ohjelmointitapa kohdan 3.4 sivulta 22.
[5]Katso *PreImage* ja *PostImage* kohdasta 3.4 sivulta 18.

hän nimeämiskäytäntöön löytyy Dynamics CRM -alustan tavasta käsitellä vedoksia. Kaikki yhteen tapahtumankäsittelylinjaston tapahtumaan liittyvät saman tyyppiset vedokset ovat samassa kokoelmassa, eli *PreImage*-vedokset omassaan ja *PostImage*-vedokset omassaan. Näitä kokoelmia indeksoidaan vedoksen nimellä. Jos kokoelmaan lisättäisiin useita saman nimisiä vedoksia eri sisällöillä, niin vain viimeisimpänä lisätty jäisi voimaan. Tällöin ohjelmistokehittäjän pyytämä vedos ei välttämättä sisälläkään ohjelmistokehittäjän olettamia tietoja. Tällainen tilanne voi aiheuttaa asiakkaan ympäristöön hyvin hankalasti jäljitettäviä virheitä.

Komponenttikehyskontekstin toteutusta, eri variaatioita ja variaatioiden keskinäisiä suhteita on kuvattu tarkemmin liitteessä D. Yhteenvetona komponenttikehyskonteksti toimii sekä fasadina että siltana. Fasadin roolissa komponenttikehyskonteksti tarjoaa kehittäjälle yhden rajapinnan useammasta eri paikasta koostetun tiedon käyttämiseen yhtenäisen rajapinnan läpi. Silta-suunnittelumallilla on puolestaan laajennettu komponenttikehyskontekstin eri variaatioita käyttämällä periytymistä sekä piilotettu konkreettisten toteutustekniikoiden erot.

Komponenttikehyskontekstin kautta voidaan tarjota uusia Dynamics CMR -järjestelmän kehitystyötä helpottavia toteutuksia sitä mukaa kun niitä tulee vastaan komponenttikehyksen elinkaaren aikana. Liitännäisten komponenttikehyskontekstissa valtaosa tehdyistä kehitystyötä helpottavista toteutuksista liittyy Dynamics CRM:n suorituskontekstin käsittelyyn. Räätälöityjen työnkulun vaiheille tarjottavassa komponenttikehyskontekstissa on vähemmän erikoistunutta toiminnallisuutta kuin liitännäisten komponenttikehyskontekstissa. Tämä johtuu enimmäkseen siitä, että liitännäisten rooli on ollut Dynamics CRM -alustassa suurempi. Liitännäisten toteutuksessa on enemmän evoluution kautta tulleita mahdollisuuksia suorituskontekstin hyödyntämiseen kuin räätälöityjen työnkulun vaiheiden toteutuksissa, ja tämä heijastuu komponenttikehyskontekstiin.

6. KONFIGURAATION HALLINTA

Komponenttikehyksen käyttöönottoon liittyy sen konfiguraation hallinta. Diplomityön toteutuksen alussa uusin Dynamics CRM -versio oli versio 2013. Diplomityön toteutuksen aikana julkaistiin versiot 2015, 2016 ja Dynamics 365 (D365)[1]. Kaikkien näiden ohjelmointirajapinta on hyvin samankaltainen. Komponenttikehyksen on tarkoitus tukea näitä kaikkia Dynamics CRM -versioita. Aiempien Dynamics CRM -versioiden julkaisuhistoriassa uusin julkaistu Dynamics CRM -versio on toistaiseksi aina tukenut myös edellisen version SDK-kirjastoja vastaan käännettyjä toteutuksia, mutta ei tätä vanhempia SDK-kirjastoja vastaan käännettyjä toteutuksia. Käytännössä siis esimerkiksi Dynamics CRM 2016 tukee version 2015 SDK-kirjastoja, mutta ei version 2013 SDK-kirjastoja. Tämä tuo komponenttikehyksen eri versioiden jakelemiseen haasteen. Tavoitteena on pitää komponenttikehys toimivana kaikille näille eri versioille ja tarjota komponenttikehyksestä samat toiminnot kaikille näille eri versioille.

Tämä tavoite käsitti käytännössä kolme suurempaa ratkaistavaa kokonaisuutta. Näistä yksi liittyi siihen, miten tarjottava toteutus on ylipäätään käytettävissä niin paikallisissa Dynamics CRM -asennuksissa kuin myös Dynamics CRM:n pilvipalvelussa. Toinen kokonaisuus puolestaan liittyi komponenttikirjaston jakelemiseen yrityksen sisäisesti tulevan käyttäjäkunnan käyttöön. Kolmas ratkaistava kokonaisuus oli se, miten eri Dynamics CRM -versioille saadaan tarjottua komponenttikehyksestä eri .NET-versioita ja eri Dynamics CRM SDK -kirjastoja vastaan käännettyt versiot. Tässä luvussa on kuvattu, miten nämä kolme kokonaisuutta ratkaistiin.

6.1 Komponenttikehyksen kirjastojen lataaminen pilviympäristöissä

Komponenttikehyksen eri osien välisten riippuvuuksien pilkkominen luvussa 5 kuvatun mukaisesti aiheutti sellaisen lähdekoodin rakenteen, josta syntyy käännettäessä

[1]Dynamics 365 (D365) on Microsoftin uudelleennimeämä Dynamics CRM. Teknisten versionumeroiden puitteissa Dynamics CRM 2013 on versio 6.0, 2015 on versio 7.0 ja 2016 on versio 8.0. D365 voi viitata sekä versioon 8.2 että 9.0. Tässä diplomityössä D365 viittaa versioon 9.0.

useita eri kirjastoja. Komponenttikehyksen käyttäjän tulee toteutuksessaan viitata jokaiseen näistä kirjastoista, mikäli niitä kaikkia tarvitaan asiakaskohtaisen logiikan toteuttamiseksi. Kaikkien viitattujen kirjastojen tulee olla saatavilla ajonaikaisesti, jotta tehty toteutus toimii kohdejärjestelmässä.

Viitattujen kirjastojen ajonaikaiseen tarjoamiseen on käytännössä kaksi eri vaihtoehtoa. Kirjastot voivat olla paikallisesti asennettuina Dynamics CRM -palvelimelle, josta käyttöjärjestelmä lataa ne ajonaikaisesti. Vaihtoehtoisesti viitatut kirjastot voidaan yhdistää yhdeksi Dynamics CRM -järjestelmän tietokantaan ladattavaksi kirjastoksi, joka sisältää myös suoritettavien liitännäisten ja työnkulun vaiheiden toteutukset. Käytännössä kirjastojen yhdistäminen tarkoittaa useiden eri kirjastojen yhdistämistä yhdeksi tiedostoksi [105].

Kirjastojen asentaminen palvelimelle voidaan tehdä joko Dynamics CRM -asennuksen hakemistoon tai palvelimen GAC-hakemistoon. GAC-hakemisto on .NET-kehyksen käyttämä laitekohtainen ja koko laitteessa saatavilla oleva hakemisto, josta voidaan ladata kirjastoja dynaamisesti [55]. Terminä GAC on lyhennys englannin kielen sanoista *Global Assembly Cache* [81]. Dynamics CRM -alustan kannalta GAC-asennustavan suurin ongelma on sen vaatima pääsy palvelimen levyjärjestelmään ja tämä puolestaan on estetty Dynamics CRM:n pilviympäristöissä. Tämä asennustapa ei siis sovellu komponenttikirjaston ajonaikaiseksi tarjoamiseksi pilviympäristöissä. GAC-asennusta voidaan kuitenkin käyttää komponenttikehyksen kirjastojen tarjoamiseen paikallisille palvelimille asennetuissa Dynamics CRM -järjestelmissä.

Palvelimen levyjärjestelmälle asentamisen sijasta komponenttikehyksen kirjastot voidaan yhdistää asiakaskohtaisesti toteutettujen liitännäisten ja työnkulun vaiheiden kanssa yhdeksi Dynamics CRM:n tietokantaan ladattavaksi kirjastoksi. Tällöin asiakastoteutuksen viittaamat komponenttikehyksen kirjastot ovat ajonaikaisesti tarjolla kyseisestä kirjastotiedostosta. Useiden .NET-kirjastojen yhdistämiseen yhdeksi kirjastoksi on tarjolla useita ilmaisia työkaluja kuten Costura [10], LibZ [28], ILMerge [105] ja ILRepack [113].

ILMerge-yhdistystyökalun käytöstä löytyi Microsoftin omasta Dynamics CRM -blogista hyvä, komponenttikehyksen käyttötapausta vastaava esimerkki vuodelta 2010 [5]. Samaiseen blogikirjoitukseen [5] päivitettiin vuonna 2015 tieto, jonka mukaan ILMerge-työkalun käyttö ei ole tuettua Dynamics CRM:n kanssa, mutta sitä ei ole myöskään estetty. Microsoft ei kuitenkaan toistaiseksi ole tarjonnut tähän muuta ratkaisua. ILMergen avulla saatiin tehtyä komponenttikehyksen kirjastojen ajonaikainen tarjoaminen asiakaskohtaisille toteutuksille. Tämän ratkaisun on todettu toimivan sekä Dynamics CRM:n pilviympäristöissä että paikallisissa asennuksissa.

Tämän takia ILMerge valikoitui komponenttikehyksen esimerkkitoteutuksen yhdistämistyökaluksi.

Yhdistämisratkaisussa on pienenä miinuspuolena se, että se pitää konfiguroida käsin jokaiseen asiakasprojektiin. Konfiguroinnissa ILMerge-työkalu asetetaan käynnistymään automaattisesti jokaisen onnistuneen lähdekoodiprojektin käännön jälkeen. Konfigurointi tulee tehdä niihin lähdekoodiprojekteihin, joiden lopputuloksena syntyy Dynamics CRM -palvelimella ajettavia liitännäisiä tai räätälöityjä työnkulun vaiheita. Ohjeet tarvittavan konfiguroinnin tekemiseen on sisällytetty komponenttikehyksen käyttöönoton ohjeisiin.

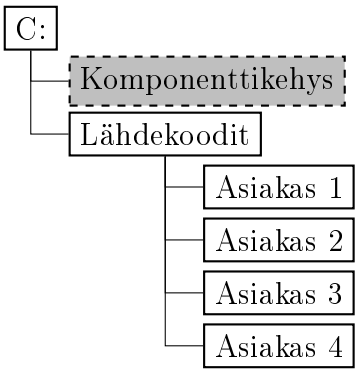
6.2 Komponenttikehyksen jakelu

Komponenttikehyksen kehitystyön edetessä piti selvittää, miten se saadaan jaeltua tehokkaasti käyttäjille. Accountor Enterprise Solutions Oy:ssä ei ollut olemassa olevia jakelutapoja sisäisten valmiskirjastojen jakelemiseen komponettikehyksen kehitystyön aikaan.

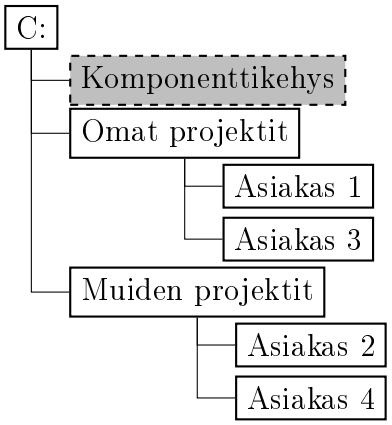
Jakeluun mietittiin useita eri tapoja. Ensimmäinen mietitty tapa oli jo kohdassa 2.2 mainittu lähdekoodin kopiointi. Tässä jakelutavassa lähdekoodi kopioitaisiin komponenttikehyksen pääkehityshaarasta osaksi asiakasprojektien lähdekoodia. Suureksi ongelmaksi koettiin liian kevyt pakottavuus komponenttikehyksen pääkehityshaaran käyttämiselle. Tämä tapa olisi voinut hidastaa päähaaran kehitystä asiakasprojektien teknisten arkkitehtien ja ohjelmistokehittäjien jalostaessa siitä omia versioitaan. Toiseksi suureksi ongelmaksi koettiin keskitetyn virheenkorjauksen puuttuminen. Virheet olisi pitänyt korjata kopioimalla komponenttikehyksen lähdekoodi aina uudestaan sen pääkehityshaarasta. Tässä yhteydessä mietittiin myös versionhallinnan kehityshaarojen haarauttamista. Tällöin komponenttikehyksen päähaarasta tehtäisiin alihaara jokaiseen sitä käyttävään asiakasprojektiin. Alihaaroihin tehtyjä muutoksia puolestaan pystyisi siirtämään eri haarojen välillä versionhallinnan yhdistämistoimintojen avulla. Alihaarojen käytössä todettiin kuitenkin olevan mukana valtaosa samoista miinuspuolista kuin lähdekoodin kopioinnissa.

Toisena vaihtoehtona mietittiin komponenttikehyksen eri lähdekoodiprojektien sijoittamista versionhallinnassa tiettyyn vakiosijaintiin ja näihin sijainteihin viitattaisiin asiakasprojekteista suhteellisten polkujen avulla. Tässä nousi suureksi käytettävyysongelmaksi eri ohjelmistokehittäjien eri käytännöt siinä, miten versionhallinnan hakemistot on kiinnitetty heidän paikallisiin hakemistoihinsa. Visual Studio -kehitysympäristö käyttää projektitiedostoissaan suhteellisia viittauksia, mikäli se

on mahdollista. Ongelmista saa käsityksen, jos oletetaan yhden kehittäjän käyttävän kuvan 6.1 mukaista paikallista hakemistorakennetta ja toisen käyttävän kuvan 6.2 mukaista hakemistorakennetta. Tällöin kumpikin kehittäjä joutuisi käyttämään erilaisia hakemistoviittauksia komponenttikehyksen lähdekoodiin. Suhteellisten polkujen käyttö aiheuttaa esimerkin mukaisessa tilanteessa ongelman, jossa yhteisestä versionhallinnasta otettu yhdellä kehittäjällä kääntyvä lähdekoodi ei käänny toisella johtuen erilaisista suhteellisista hakemistoviittauksista. Ongelmia versionhallinnasta otetun lähdekoodin kääntymisessä haluttiin välttää.



Kuva 6.1 *Kaikkien asiakkaiden lähdekoodi samassa juurihakemistossa.*



Kuva 6.2 *Asiakkaiden lähdekoodit eri juurihakemistoissa.*

Lisäksi suora viittaaminen komponenttikehyksen lähdekoodiin todennäköisesti aiheuttaisi ongelmia komponenttikehyksen julkisen rajapinnan kehityksen kanssa. Tässä tavassa asiakasprojektien lähdekoodi käyttäisi aina uusinta versionhallinnassa olevaa komponenttikehyksen versiota. Tämän takia jokainen lisäys komponenttikehyksen rajapintoihin heijastuisi välittömästi sivuvaikutuksena jokaiseen asiakasprojektiin. Tästä sivuvaikutuksesta ei myöskään saisi mitään palautetta komponenttikehyksen muutosten yhteydessä lähdekoodia käännettäessä. Asiakasprojektien kokemat sivuvaikutukset voivat olla jossakin tapauksessa hyvin suuret, jos asiakasprojekti viittaa suoraan komponenttikehyksen lähdekoodiin.

Kolmas ajatus komponenttikehyksen jakelumahdollisuuksista oli pystyttää yrityksen sisäinen NuGet-palvelu, josta komponenttikehystä voitaisiin jaella käyttäjille binäärimuotoisena. NuGet on Visual Studio -kehitysympäristön mukana tuleva työkalu, joka mahdollistaa ulkoisten kirjastojen helpon käyttöönoton ja hallinnoinnin lähdekoodiprojekteihin [20]. Kirjastojen asentamisen yhteydessä NuGet lisää kirjastoviittaukset automaattisesti lähdekoodiprojektiin. Lisäksi tässä yhteydessä voidaan tarvittaessa kopioida lähdekoodiprojektiin paketissa mukana olevia tiedostoja, kuten esimerkiksi lähdekoodia tai kuvia. NuGet-työkalu mahdollistaa kirjastojen asentamisen lisäksi tarjolla olevien valmiskirjastojen etsimisen ja muun muuassa kirjastojen

riippuvuuksien automaattisen hallinnan sekä automaattiset päivitysten tarkistukset lähdeprojektiin asennettuihin kirjastoihin. Lyhyesti, NuGet on tapa helpottaa lähdekoodiprojektin käyttämien ulkoisten kirjastojen käyttöönottoa ja vähentää niiden ylläpitoon kuluvaa aikaa. NuGet-palvelun kautta jaelluista kirjastoista käytetään nimitystä NuGet-paketti. Näitä paketteja pystyy tekemään NuGet-työkalun avulla haluamistaan lähdekoodiprojekteista. Lisäksi näitä itse tehtyjä paketteja voidaan jaella esimerkiksi yrityksen sisällä perustamalla tätä varten oma NuGet-palvelu.

Paketoimalla komponenttikehys NuGet-paketin sisälle ja jakelemalla sitä yrityksen sisäisen NuGet-palvelun kautta, saadaan vältettyä edellisissä kappaleissa kuvattuja ongelmia lähdekoodien kopioinnin, suhteellisten polkujen sekä rajapinnan versioinnin kanssa. NuGet-jakelun epäiltiin vähentävän myös mahdollisuuksia rinnakkaisten kehityshaarojen syntymiseen jakelun tapahtuessa binäärimuodossa. Lisäksi tämän jakelutavan mietittiin mahdollistavan tulevaisuudessa myös muiden hyödyllisten, omaa kehitystä olevien työkalujen jakelun yrityksen sisällä.

Visual Studio -kehitysympäristön projektityyppien mukana voidaan tuoda ohjelmistokehittäjien käyttöön erilaisia valmiita pohjia, joiden avulla he voivat lähteä laajentamaan projektejaan [47]. Tällaisen projektityypin kehittämistä Visual Studio -kehitysympäristöön mietittiin neljäntenä vaihtoehtoisena tapana komponenttikehyksen jakeluun. Tämän avulla saataisiin joitakin samoja hyötyjä kuin NuGet-palvelun avulla jakamisesta. Tämän tavan todettiin kuitenkin olevan melko harvinainen kirjastojen jakelemiseksi. Lisäksi haasteena pidettiin ylläpitoa tulevien Visual Studio -versioiden kanssa. Accountor Enterprise Solutions Oy:ssä oli kokemuksia siitä, etteivät kaikki Visual Studion räätälöidyt projektityypit enää auenneet seuraavissa Visual Studio -versioissa.

Näiden vaihtoehtojen pohjalta päädyttiin NuGet-pohjaiseen jakeluun. Tämän tavan tärkeimmät edut olivat helppo käyttö työkalun tullessa Visual Studio -kehitysympäristön mukana, riippuvuuksien automaattinen hallinta ja mahdollisuus sulkea komponenttikehys sen lähdekoodiin tehtäviltä muutoksilta. NuGet-palvelu tulee sisäiseen käyttöön ja aluksi sieltä jaellaan ainoastaan komponenttikehystä. Tulevaisuudessa NuGet-palvelun kautta tullaan todennäköisesti jakelemaan myös muita yrityksen omia, yleiskäyttöisiä toteutuksia.

6.3 Eri Dynamics CRM -versioiden tukeminen

Komponenttikehyksen toiminnallisuudet haluttiin pitää samoina kaikissa komponenttikehyksen tukemissa Dynamics CRM -versioissa. Lisäksi haluttiin julkaista erillinen versio jokaista tuettua Dynamics CRM -versiota kohden. Käytännössä vaa-

timuksena oli siis komponenttikehyksen jokaisen version julkaisu Dynamics CRM 2013, 2015 ja 2016 sekä D365 SDK-kirjastoja vastaan. Tässä piti huomioida myös näiden Dynamics CRM SDK -kirjastojen erilaiset minimivaatimukset .NET-kirjaston suhteen.

Toistaiseksi komponenttikehyksen toteutuksissa ei ole muita eroja eri Dynamics CRM -versioiden kesken kuin riippuvuudet Dynamics CRM:n SDK-kirjaston eri versioista. Tavoitteena oli saada aikaan tuki eri Dynamics CRM -versioille niin, että lähdekoodin variaatiot olisivat mahdollisimman pieniä. Ratkaisuksi löytyi lähdekoodiprojektitasolla tehtävä ehdollinen MSBuild-käännösmoottorin konfigurointi [70; 15]. Konfiguroinnissa päädyttiin ratkaisuun, jossa komponenttikehyksen lähdekoodin voi valita käännettäväksi joko Dynamics CRM 2013, 2015, 2016 tai D365 -versiota vastaan. Näistä jokainen konfiguraatio on muokattu mahdollistamaan kääntö tuotantooptimoituna tai debug-symbolien kanssa [96].

Lähdekoodin kääntämisen lisäksi eri Dynamics CRM -versioiden tukeminen vaatii kullekin versiolle omat NuGet-paketit. NuGet-paketit päädyttiin julkaisemaan kunkin Dynamics CRM -kohdeversion sisäisen versionumeron mukaan. Esimerkiksi Dynamics CRM 2013 -version sisäinen versionumero on 6.x ja version 2015 sisäinen versionumero on 7.x. Näitä vastaavasti Dynamics CRM 2016 -versiolle tarkoitetut komponenttikehyksen NuGet-paketit ovat versionumeroltaan 8.x. Näin kaikkien pakettien yksilölliset nimet pysyvät samana. Tämä helpottaa myös versiopäivityksissä tehtävää komponenttikehyksen NuGet-pakettien päivittämistä.

6.4 Komponenttikehyksen konfigurointi NuGet-asennuksesta

Kun käyttäjä ottaa komponenttikehyksen käyttöön, hänelle tarjotaan kahta eri NuGet-jakelupakettia. Toinen jakelupaketti sisältää pelkät komponenttikehyksen kirjastot, kun taas toinen paketti sisältää myös esimerkkikonfiguraatiot. Näillä kahdella eri jakelupakettitavalla on eri tavoitteet. Kun komponenttikehys tarjotaan esimerkkikonfiguraation kanssa, käyttöönotto nopeutuu ja käyttöönoton kynnys madaltuu. Ilman esimerkkikonfiguraatiota tarjottava jakelu on tarkoitettu tilanteisiin, joissa tarvitaan hienojakoisempaa kontrollia konfiguraation yli. Lisäksi, joissakin tilanteissa .NET-kääntäjä vaatii lähdekoodissa viitattujen kirjastojen viittausten olevan saatavilla käännösaikaisesti, jotta viittausgraafi saadaan tarkistettua. Myös tällaisessa tilanteessa voidaan käyttää pelkkiä komponenttikehyksen kirjastoja. Automaattisten yksikkötestien toteuttaminen asiakasprojektiin on tyypillinen esimerkki tästä tilanteesta.

Komponenttikehys jaellaan useassa eri NuGet-jakelupaketissa. Näistä jokaiseen on konfiguroitu niiden vaatimat riippuvuudet muihin NuGet-paketteihin. Vaadittujen Dynamics CRM -kirjastojen osalta viitataan Microsoftin jakelemiin NuGet-paketteihin. Jakelupakettien osittelu ja suhteet pohjautuvat kohdassa 5.1 kuvattuun komponenttikehyksen kirjastorakenteeseen. Eri NuGet-jakelupaketeille yhteistä on se, että ne kaikki lisäävät automaattisesti kääntäjän vaatimat kirjastoviittaukset asiakasprojektin lähdekoodiin. Käyttäjä voi siis luottaa lähdekoodiprojektinsa kääntyvän ilman mitään manuaalitoimenpiteitä sen jälkeen, kun hän on lisännyt siihen viittauksen komponenttikehyksen NuGet-paketteihin.

Esimerkkikonfiguraation sisältävässä NuGet-paketissa lähdekoodiprojektiin lisätään automaattisesti valmiiksi konfiguroidut esimerkit sekä liitännäisiä että räätälöityjä työnkulun vaiheita varten. Näiden lisäksi omana lähdekooditiedostona tulee esimerkki pakollisista erikoistamisrajapinnan toteutuksista. Esimerkkikonfiguraatio sisältää räätälöimättömästä Dynamics CRM -instanssista muodostetun DAO-kontekstiluokan, siinä käytetyt DTO-luokat ja tarvittavan esimerkin ILMerge-työkalun käytöstä. Esimerkkikonfiguraatio tarjoaa komponenttikehyksestä vain staattisesti sidotun version. Esimerkkikonfiguraation tarjoamisella halutaan ohjata Dynamics CRM -kehitystyötä enenevissä määrin staattisesti sidottuun kehitysmalliin.

Esimerkkikonfiguroidun jakelupaketin asennuksen jälkeen käyttäjä periyttää liitännäiset luokasta *CustomerSpecificPluginBase* ja räätälöidyt työnkulun vaiheet luokasta *CustomerSpecificWorkflowStepBase*. Käyttäjän on tarkoitus nimetä nämä kantaluokkien esimerkkitoteutukset vastaamaan kyseisen projektin nimeämiskäytäntöjä. Komponenttikehyksen dokumentaatio on tarjolla Visual Studion IntelliSense-kommenttien avulla. Tämä dokumentaatio asentuu NuGet-jakelupakettien mukana. Lisäksi komponenttikehyksestä on yrityksen sisäinen, yleisempi käyttöönoton dokumentaatio.

7. KEHITYSPROSESSI

Komponenttikehyksen kehitysprosessi jakautuu kolmeen eri kokonaisuuteen. Toteutuksen etenemisprosessin ydinkohdat muodostavat näistä ensimmäisen. Toinen kokonaisuus muodostuu siitä, miten etenemisprosessin aikana on varmistettu soveltuvuus aiottuun käyttötarkoitukseen. Kolmas kokonaisuus on komponenttikehyksen käyttöönotto asiakasprojektien toteutuksissa.

7.1 Toteutusprosessi

Tämän diplomityön voi katsoa alkaneen vuonna 2013. Olin tuolloin heinäkuusta lokakuuhun yhtenä ohjelmistokehittäjänä eräässä isossa projektissa F. Kyseisen projektin tekninen arkkitehti oli tehnyt projektiin yhteisen kantaluokan kaikille liitännäisille. Pidin tätä hyvin kustannustehokkaana tapana liitännäisille yhteisten asioiden esittelemiseksi ja toteuttamiseksi. Kiinnitin huomiota erityisesti siihen, ettei samoja asioita tarvinnut tällöin toteuttaa aina erikseen, vaan työajan sai käytettyä tuottavammin. Tässä kyseisessä kantaluokassa oli yhtenäistetty lokitusta ja loppukäyttäjälle asti näkyviä virheilmoituksia. Koin nämä kaikki hyvinä asioina.

Kun minut vuoden 2013 syyskuussa kiinnitettiin tekniseksi arkkitehdiksi erääseen isohkoon projektiin D, halusin hyödyntää edellä mainittuja, hyviksi käytännöiksi kokemiani asioita. Toteutin heti projektin D alussa kantaluokan liitännäisiä varten. Tämän projektin alkuvaiheessa tiedostettiin tarve toteuttaa merkittävissä määrin toiminnallisuutta myös räätälöityinä työnkulun vaiheina. Toteutin kantaluokan myös räätälöidyille työnkulun vaiheille. Halusin projektiin vain yhden toteutuksen liitännäisten ja työnkulkujen yhteisesti käyttämistä huolenaiheista. Näitä olivat lokitus, virheenkäsittely sekä ajonaikaisen kontekstin tarjonta. Toteutin nämä yhteiset huolenaiheet yhteen luokkaan, jota käytettiin sekä liitännäisten että räätälöityjen työnkulun vaiheiden kantaluokissa.

Näiden kokemusten innoittamana lähdin toteuttamaan komponenttikehystä silloisen Mepco Oy:n CRM-yksikön käyttöön. Komponenttikehyksen idea esiteltiin yrityksen sisäisessä kehittäjäpäivässä tammikuussa 2014. Tässä yhteydessä käytiin keskustelua tärkeiksi koetuista toiminnallisista ominaisuuksista. Keskustelua tuli paljon

myös käyttöönoton helppouden tärkeydestä. Lisäksi keskustelussa nostettiin esille ylläpitonäkökulma, sillä komponenttikehyksessä olevat virheet päätyisivät kaikkiin sitä käyttäviin asiakastoteutuksiin. Tämän tilaisuuden keskustelut vaikuttivat paljon luvussa 4 kuvattuihin vaatimuksiin ja suunnittelun lähtökohtiin.

Kehittäjiltä ei tullut tämän tilaisuuden jälkeen kuin muutama toive komponenttikehykseen liittyen, vaikka niitä oli varta vasten pyydetty. Nämä liittyivät enimmäkseen käyttötuessa saatuihin kokemuksiin siitä, miten asiakkaan tuotantoympäristössä voidaan helpommin selvittää ongelmatilanteita. Käytännössä nämä vaikuttivat virhetilanteiden hallinnan oletustoteutukseen ja kirjoitettavan lokiviestin muotoiluun helposti hyödynnettäväksi.

Keskustelin tavoiteltavista vaatimuksista myös silloisen esimieheni kanssa. Hän puolestaan keskusteli liiketoiminnan tavoitteista kollegoidensa kanssa. Esimieheni kanssa käydyt keskustelut antoivat tälle työlle viitekehyksen liiketoiminnan näkökulmasta tuleville ei-teknisille tavoitteille.

Diplomityön tuloksena syntyneen komponenttikehyksen ensimmäinen käyttökerta tehtiin asiakasprojektissa E loppukeväällä 2014. Tässä yhteydessä oli tehty tuki liitännäisille. Räätälöityjen työnkulun vaiheiden tuki lisättiin kesän 2014 aikana. Saman vuoden syksyllä komponenttikehys otettiin käyttöön uudessa asiakasprojektissa V. Tämä oli yrityksemme mittakaavassa iso projekti. Komponenttikehyksen arkkitehtuuri oli tässä kohtaa muotoutunut malliin, josta sitä ei ollut enää tämän jälkeen tarvetta muokata.

Tammikuun 2014 sisäisessä kehittäjäpäivässä oli noussut esille toteutuksen saaminen automaattisen testauksen pariin. Keskityin tähän asiaan komponenttikehyksen rakenteen muotouduttua projektin V aikana julkaisuvalmiuden rajalle. Yrityksessä ei ollut aiemmin tehty automaattista yksikkötestausta Dynamics CRM -tuotteen kanssa. Tämä vaati paljon selvitystyötä niin automaattisesta yksikkötestauksesta ylipäätään kuin myös käyttöön sopivista kirjastoista. Valittujen kirjastojen käyttöönotto paljastui melko helpoksi ottaen huomioon sen, ettei minulla ollut yhtään ennakkoosaamista yksikkötesteistä. Automaattisten yksikkötestien toteutuksen jälkeen komponenttikehykseen ei ole ollut tarvetta tehdään suuria toiminnallisia muutoksia tai lisäyksiä. Profilointituen lisääminen alkukesällä 2015 on ollut ainoa suurempi uusi toiminnalinen kokonaisuus.

7.2 Suunnittelun ja toteutuksen varmentaminen

Komponenttikehyksen vaatimusten keräämisen jälkeen kehittämistä jatkettiin projektissa D. Käytännössä kyseisessä projektissa tehtiin komponenttikehyksen prototyyppi. Tässä yhteydessä saatiin hyvä käsitys siitä, mitkä huolenaiheet on syytä eriyttää komponenttikehyksen vastuulle. Lisäksi havaittiin paljon riippuvuusongelmia, koska kaikki asiakastoteutukset riippuivat konkreettisista toteutuksista. Näistä prototyyppivaiheen kokemuksista tärkeimmäksi nousivat huolenaiheiden abstrahointi rajapintojen avulla sekä näiden huolenaiheiden tarjoaminen ohjelmistokehittäjille komponenttikehyskontekstin avulla.

Muutama kuukausi prototyyppivaiheen jälkeen komponenttikehyksen toimintaa testattiin asiakasprojektissa E. Testauksella haluttiin saada vahvistus komponenttikehykseen valitun julkisen toteutuksen toimivuudesta. Tämän asiakasprojektin kehitystyöhön osallistui lisäkseni kaksi muuta ohjelmistokehittäjää. Heidän kommenttiensa mukaan silloinen komponenttikehyksen rakenne ei ollut ohjelmistokehittäjän näkökulmasta huono.

Projektin V yhteydessä komponenttikehyksen toimivuutta saatiin varmistettua huomattavasti kattavammin. Suurimmat tänä aikana esille nousseet muutostarpeet liittyivät sovelluslogiikan ympäristökohtaiseen parametrisointiin asetustietueiden avulla sekä loppukäyttäjille näytettävien virheilmoitusten käsittelyyn. Havaittiin, että asetustietueiden, kuten käytännössä kaikkien muidenkin komponenttikehyksen käsittelemien järjestelmän aspektien, on syytä olla abstrahoitu roolirajapintojen taakse. Parametristojen käsittelyn osalta tuli myös tarve käyttää samaa parametrisointirajapintaa komponenttikehyksen ulkopuolisessa sovelluslogiikassa. Esimerkkinä tällaisesta ovat muun muuassa erilaiset komentorivisovellukset.

Virheilmoitusten näyttämisestä loppukäyttäjille ja tähän liittyvästä arkkitehtuurista käytiin paljon sisäistä keskustelua projektin V aikana. Näiden keskustelujen lopputuloksena komponenttikehyksen poikkeuskäsittelyyn syntyi ratkaisu, jossa virheenkäsittelijää voidaan vaihtaa asiakasprojektikohtaisesti kohdan 5.2 mukaisesti. Todettiin, että toteutettu ratkaisu mahdollistaa myös osittain tunnistetun tarpeen virheilmoitusten kääntämiselle.

Komponenttikehyksen koko julkiseen rajapintaan tehtiin automaattiset yksikkötestit. Automaattisen yksikkötestauksen osalta päädyin käyttämään testauskirjastoa nimeltä xUnit [114] tekemäni kirjastovertailun jälkeen. Testauskirjaston lisäksi yksikkötesteihin tarvittiin simulointikirjasto esittämään testauksen kohteena olevan koodin taustalleen vaatimaa infrastruktuuria. Esimerkki tällaisesta infrastruktuurin simuloinnista on Dynamics CRM:n liitännäiselle tarjoama kontekstirajapinta. Simu-

lointikirjastoksi valitsin NSubstitute-kirjaston [100]. Komponenttikehyksestä löytyi yksikkötestien avulla bugeja heti ensimmäisenä yksikkötestien toteutuspäivänä. Automaattisen yksikkötestauksen kannalta haastavimmaksi osoittautui komponenttikehyksen profilointiominaisuuden testaaminen, koska tällöin Dynamics CRM:n web service -rajapinnan toiminta tuli ensimmäistä kertaa simuloitavaksi.

7.3 Käyttöönotto

Yrityksessämme ei ollut kokemusta valituksi jakelukanavaksi tulleesta NuGet-pakettien jakelusta tai paketoinnista. Tutustuin ensin paketointiin. Tässä yhteydessä tuli ratkaistavaksi pakettien nimeämiseen, riippuvuuksien hallintaan ja asiakasprojekteihin lisättävään esimerkkiaineistoon liittyviä asioita. Lisäksi ratkaistavaksi tuli niin asiakasprojektien elinkaaren aikaiseen ylläpidettävyyteen kuin käytöönoton helppouteen liittyviä asioita. Lisäksi hoidin tarvittavan NuGet-jakeluinfrastruktuurin perustamisen yrityksemme sisälle.

Muutama yrityksemme ohjelmistokehittäjä testasi komponenttikehyksen käyttöönottoa toisella toimistollamme syksyllä 2015. He eivät tunteneet komponenttikehystä aiemmin. Testaus tehtiin suunnittelemattomasti muutaman päivän varoitusajalla siellä käynnistyvän uuden asiakasprojektin yhteydessä. Huolimatta projektin teknisen arkkitehdin aikomuksista, kyseisessä projektissa nousi muutosvastarintaa liian hankalaksi koetun käyttöönoton myötä. Projektin tekninen arkkitehti harkitsi komponenttikehyksen ottamista käyttöön toisen teknisen arkkitehtimme suosituksesta. Eräs muutosvastarinnan ilmentymä oli kilpailevan komponenttikehysratkaisun kehitystyön aloittaminen tässä asiakasprojektissa.

Yksi kehitysidea tästä testauksesta oli valmiin DAO-kontekstiluokan ja sen päivitysskriptin lisääminen NuGet-jakelupakettiin. Tällä haluttiin poistaa käsin tehtävää konfigurointityötä ja siten lisätä komponenttikehyksen houkuttelevuutta. Tämän muutoksen jälkeen käyttöönottoa testattiin uudelleen. Tällöin suurimmaksi koetuksi ongelmaksi nousi koodiprojektin kääntymättömyys sen jälkeen kun komponenttikehys oli asennettu NuGet-paketista. Kun tämä ongelman oli ratkaistu NuGet-paketin konfiguraatiomuutoksella, komponenttikehyksen ensimmäinen virallinen versio julkaistiin joulukuussa 2015. Komponenttikehyksen jakelupakettien osalta ei ole tarvinnut tehdä rakennemuutoksia enää tämän jälkeen.

Käyttöönottoja on tapahtunut komponenttikehyksen julkaisun jälkeen sekä uusissa että vanhoissa asiakasprojekteissa. Vanhoissa asiakasprojekteissa komponenttikehyksen käyttöönotto on tapahtunut versiopäivityksen yhteydessä, kun taustalla

oleva Dynamics CRM -järjestelmä on päivitetty uudempaan versioon. Komponenttikehyksen käyttöönottoa tukemaan on kirjoitettu tarkistuslistatyyppinen ohjeistus asioista, jotka pitää tehdä tai huomioida. Käyttöönoton jälkeisen ensimmäisen vuoden aikana esiintyi jonkin verran pienemmän mittakaavan muustosvastarintaa. Tämä ilmeni lähinnä joidenkin kehittäjien kielteisenä suhtautumisena muilta tulevan toteutuksen käyttöön, eli niin kutsutun Not-Invented-Here -syndroomaan [23] ilmentyminä.

8. TULOSTEN TARKASTELU

Diplomityössä havaitaan olevan selvästi kaksi osaa. Näistä toinen on syntyvän komponenttikehyksen arkkitehtuuri. Toinen osa on jaeltavan lopputuloksen konfiguraation hallinta. Näitä asioita tarkastellaan tässä luvussa eri näkökulmista. Kohdassa 8.1 komponenttikehyksen toteutusta peilataan yleisiä ohjelmistoarkkitehtuurien periaatteista vastaan. Kohdassa 8.2 komponenttikehystä verrataan ohjelmistokehyksistä esitettyyn teoriaan. Komponenttikehyksen toteutuksen taustalla oli joukko Accountor Enterprise Solutions Oy:n liiketoiminnasta kummunneita asioita. Tehdystä toteutuksesta saatuja hyötyjä peilataan näihin liiketoiminnan tarpeisiin kohdassa 8.3. Komponenttikehyksellä oli myös joukko muita kuin liiketoiminnasta kumpuavia vaatimuksia ja näitä tarkastellaan kohdassa 8.4.

8.1 Toteutuneen arkkitehtuurin tarkastelu

Komponenttikehyksessä on käytetty roolirajapintoja [27, s. 77] erityisesti asiakaskohtaisen konfiguraation erikoistamisen (kohta 5.2) ja komponenttikehyskontekstin (kohta 5.4) yhteydessä. Roolirajapintojen kuvaamisessa on pyritty kohdassa 3.1 esitetyn mukaisesti tunnistamaan komponenttikehyksessä esiintyvät roolit ja tekemään niiden pohjalta funktionaalinen dekompositio. Roolirajapintojen tunnistamisen voi katsoa olevan onnistunut, sillä esiteltyjen roolirajapintojen määrää tai niiden vastuita ei ole tarvinnut muuttaa komponenttikehyksen varhaisten testiversioiden jälkeen.

Asiakaskohtaisen konfiguraation erikoistamisen yhteydessä roolirajapintoja käytetään komponenttikehyksen rakentajan parametreissä sekä niiden vaatimuksiin tarjottavissa oletustoteutuksissa. Komponenttikehyskonteksti puolestaan toimii useammassa, rajapintojen avulla erotetussa roolissa. Monet näistä rooleista tarjoavat käyttäjille edelleen kutsuttavaksi muita toteutuksia, joista osa on kapseloitu roolirajapintojen taakse tai joiden paluuarvona on jonkin roolirajapinnan toteutus.

Komponenttikehyskontekstin osalta on havaittavissa myös kohdassa 3.1 mainittua tiedon abstrahoinnin pohjalta tehtyä dekompositiota. Tiedon piilottamista on esimerkiksi se, miten komponenttikehyskonteksti alustetaan ajonaikaisesti Dynamics

CRM:n kohdassa 3.4 kuvatun suorituskontekstin päälle ja se, miten komponenttikehyskonteksti tekee käyttäjän puolesta esimerkiksi kohdassa 5.4 kuvattuja operaatioita Dynamics CRM:n suorituskontekstin sisältöön. Komponenttikehyskontekstin kanssa on näin saatu poistettua käyttäjiltä tarvetta toistaa samaa toteutusideaa eri asiakasprojekteissa kussakin hieman eri tavalla. Lisäksi komponenttikehyskonteksti suojaa samalla toteutustaan piilottamalla käyttäjiltä konkreettisen toteutuksen.

Komponenttikehyksen toimintaa voi laajentaa tässä diplomityössä kuvatusta kolmella eri tavalla: Asiakaskohtaisten kantaluokkien laajentaminen uusilla toiminnallisuuksilla, asiakaskohtaisten miniohjelmistokehysten toteutus asiakaskohtaisen logiikan laajennuskohtaan sekä komponenttikehyskontekstin laajentaminen. Asiakaskohtaisiin kantaluokkiin voidaan kirjoittaa uutta toiminnallisuutta ja niistä voidaan periyttää uusia asiakaskohtaisia kantaluokkia johonkin tiettyyn käyttötarpeeseen. Tämän laajennusmekanismin toteutustapa on käytännössä sama kuin kohdassa 5.2 kuvattu asiakaskohtaisen konfiguraation erikoistaminen. Asiakaskohtaiseen kantaluokkaan kirjoitetaan kyseisen asiakasjärjestelmän toteutuksessa auttavia metodeja tai uusia rakentajia injektoimaan mahdollisia uusia riippuvuuksia. Lisäksi voidaan toteuttaa sekä uusia metodeja että rakentajia. Käytännön esimerkki tällaisesta voi olla kantaluokka tietojärjestelmien välisten liittymien toteutuksen helpottamiseksi. Tähän kantaluokkaan voitaisiin injektoida ulkoisen web service -rajapinnan kanssa kommunikoiva palvelu sekä kirjoittaa metodeja tämän rajapinnan kutsumisen helpottamiseksi konkreettisia integraatioita toteuttavista lapsiluokista. Komponenttikehyksen kantaluokan laajennettavuuden voidaan tältä osin todeta olevan kohdassa 3.1 kuvatun Meyerin [37] avoin-suljettu -periaatteen mukainen.

Myös komponenttikehyksen käyttämä asiakaskohtaisen logiikan erikoistaminen (kohta 5.3) ja käyttäjälle tarjottava komponenttikehyskonteksti (kohta 5.4) ovat Meyerin [37] avoin-suljettu -periaatteen mukaisesti laajennettavia. Komponenttikehyskontekstia laajennetaan esittelemällä uusi, komponenttikehyskontekstin rajapinnasta periytyvä rajapinta sekä toteuttamalla tämä rajapinta. Jotta tällainen laajennettu komponenttikehyskonteksti saadaan käyttöön, pitää laajentaa myös asiakaskohtaisen logiikan erikoistamista. Tämä tehdään toteuttamalla komponenttikehyksen kantaluokan vaatima *Run()*-metodi asiakaskohtaisen järjestelmän kantaluokassa ja esittelemällä uusi metodi asiakaskohtaisen logiikan erikoistamiseen. Edellä kuvattua komponenttikehyskontekstin laajentamisen toteutusta on havainnollistettu ohjelmalistauksessa 8.1. Käytännön esimerkki tällaisesta laajennustavasta on edellisessäkin kappaleessa käytetty tietojärjestelmien välisten liittymien toteuttamiseen tehty asiakaskohtainen kantaluokka. Tällaisessa kantaluokassa kaikkien integraatioiden yhteinen logiikka voidaan toteuttaa komponenttikehyksen vaatimaan *Run()*-metodiin. Tämän luokan sisällä voidaan vaatia lapsiluokkia toteuttamaan logiikan

integraatiokohtaiset osat. Näille metodeille voidaan antaa parametrina integraatioiden tarpeisiin laajennettu komponenttikehyskonteksti. Käytännössä tämä esimerkki mukailee Gamman [14, s. 325] esittelemää *template method* -suunnittelumallia.

```
public abstract class CustomerDAOPluginBase:
  DAOPluginBase<DAOContext>, IPlugin {

  // Rakentaja ei ole oleellinen tämän esimerkin kannalta
  public CustomerDAOPluginBase(...): base(...) {}

  // Tässä metodissa toteutetaan asiakaskohtaisen logiikan
  // erikoistamisen laajentaminen ja sulkeminen muutoksilta.
  public sealed override void Run(
    IPluginLocalDAOContext<DAOContext> execCtx) {
    // Laajennettu komponenttikehyskontekti instantioidaan.
    var extExecCtx = new CustomerExecutionContext(execCtx);
    RunWithExtended(extExecCtx);
  }

  // Periytetyt luokat erikoistavat logiikan tässä metodissa.
  public abstract RunWithExtended(
    ICustomerExecutionContext extExecCtx);

  private class CustomerExecutionContext:
    ICustomerExecutionContext {

    public CustomerExecutionContext(
      IPluginLocalDAOContext<DAOContext> execCtx) { ... }
    // Rajapinnan ICustomerExecutionContext kuvaamien
    // laajennosten toteutukset ovat täällä.
  }
}

public interface ICustomerExecutionContext:
  IPluginLocalDAOContext<DAOContext> {
  // Täällä esitellään halutut rajapinnan laajennokset.
}
```

Ohjelma 8.1 *Esimerkki miten sekä asiakaskohtaisen logiikan erikoistamista että komponenttikehyskontekstia voidaan laajentaa kohdassa 3.1 kuvatun Meyerin [37] avoin-suljettu -periaatteen mukaisesti.*

Edellä kuvatun laajennettavuuden lisäksi Meyerin [37] avoin-suljettu -periaate vaatii moduulien sisäisten toteutusten suojaamista ulkoisia muutoksia vastaan. Komponenttikehyksen asiakaskohtaisen logiikan erikoistamisen osalta suojaus on tehty kieltämällä metodin toteutuksen ylikirjoitus. Paikoitellen toteutettua metodia ei ole esitelty virtuaalisena, mikä estää toteutuksen ylikirjoituksen. Virtuaalisten ja abstraktien metodien yhteydessä on käytetty .NET-kielien *sealed*-avainsanaa. Tämä avainsana mahdollistaa luokan tai luokan tiettyjen metodien suojaamisen niin, että tehtyä toteutusta ei voi ylikirjoittaa [91]. Esimerkki tämän avainsanan käytöstä on ohjelmalistauksessa 8.1, jossa metodin *Run()*-toteutus on suojattu muutoksia vastaan. Komponenttikehyksen toteutuksessa on suojattu kohdissa 3.4 ja 5.3 mainittu, Dynamics CRM -alustan tapahtumakäsittelylinjaston kutsuma *Execute()*-metodi.

Komponenttikehyskontekstin suojaaminen ulkoisia muutoksia vastaan on tehty rajapintojen avulla. Komponenttikehyksen käyttäjältä on piilotettu ne luokat, jotka toteuttavat kohdassa 5.4 kuvatut komponenttikehyskontekstin rajapinnat. Piilotus on tehty käyttämällä luokkien esittelyssä sopivaa avainsanaa rajoittamaan näkyvyys ainoastaan toteuttavan kirjaston sisään. Koska näitä luokkia ei paljasteta komponenttikehyksen käyttäjälle, komponenttikehyskontekstin toteutus on suojattu ulkoisilta muutoksilta. Myös komponenttikehykseen injektoitavien riippuvuuksien oletustoteutukset on suojattu vastaavilla tavoilla ulkoisia muutoksia vastaan. Edellä kuvatut avoin-suljettu -periaatteen toteutumista käsittelevät kappaleet huomioiden komponenttikehyksen voi katsoa noudattavan hyvin kohdassa 3.1 kuvattua avoin-suljettu -periaatetta.

Komponenttikehyksen riippuvuuksien injektointi on toteutettu rakentajien kautta. Komponenttikehykseen rakentajien kautta injektoitavat riippuvuudet ovat luonteeltaan sellaisia, joita ilman komponenttikehys ei pysty toteuttamaan kaikkia tehtäviään. Kohdassa 3.3 on mainittu suositus rakentajan käytöstä ensisijaisena injektointitapana varsinkin, jos rakennettava instanssi ei pysty toimimaan ilman näitä riippuvuuksia. Tämä osuu yhteen komponenttikehykseen toteutetun injektoinnin kanssa. Myös muut kohdassa 3.3 mainitut riippuvuuksien injektoinnin suunnitteluperiaatteet ovat linjassa komponenttikehyksen toteutuksen kanssa.

Käänteisen kontrollin osalta komponenttikehys ei ota kovinkaan paljon kantaa sovelluksen kontrolliin. Kohdassa 3.3 mainitun mukaisesti komponenttikehys kutsuu asiakaskohtaisten toteutusten tapahtumakäsittelijöitä, eli kohdassa 5.3 kuvattua metodia *Run()*. Lisäksi komponenttikehys ohjaa asiakaskohtaisen logiikan kontrollia käytännössä juuri ennen ja jälkeen *Run()*-metodin kutsumista varmistaakseen esimerkiksi tiettyjen asioiden lokiin kirjoittamisen aina samalla tavalla.

Yhteenvetona voi todeta komponenttikehyskontekstin arkkitehtuurin hyödyntävän monia kohdassa 3.1 mainittuja, ohjelmistojen ylläpidettävyyden parantamiseen tähtääviä tekniikoita, kuten käänteinen kontrolli [12], riippuvuuksien injektointi [108], käänteisten riippuvuuksien periaate [32] ja roolirajapinnat [27]. Näiden lisäksi komponenttikehyksessä arkkitehtuuri mahdollistaa hyvän laajennettavuuden. Yhdessä nämä asiat auttavat käyttämään komponenttikehystä monen eri tyyppisissä asiaskasprojekteissa ilman, että komponenttikehyksestä syntyy rajoitteita kehitystyöhön.

8.2 Toteutuneen ohjelmistokehyksen tarkastelu

Diplomityön tuloksena syntyneen komponenttikehyksen erikoistaminen tuottaa komponentin eikä itsenäistä sovellusta. Tämä takia diplomityön tuloksena on syntynyt nimenomaan ohjelmistokehystä suppeampi komponenttikehys [27]. Toteutettu komponenttikehys kutsuu asiakaskohtaista liiketoimintalogiikkaa käänteisen kontrollin avulla kohdassa 5.3 selitetyn mukaisesti. Tältä osin komponenttikehys noudattaa ohjelmistokehysten tyypillistä, kohdassa 3.3 kuvattua käytöstä.

Toteutettu komponenttikehys tukee käyttäjiään tarjoamalla heille kohdassa 5.4 esitetyn komponenttikehyskontekstin kautta yleisiä Dynamics CRM -järjestelmän kehitystyössä käytettäviä, toteutusta helpottavia rajapintoja ja metodeja. Komponenttikehyskontekstin ja sen roolirajapintojen modulaarisuuden on havaittu jo parantaneen varsinkin toimitettujen asiakastoteutusten ylläpidettävyyttä, koska eri ohjelmistokehittäjät ovat alkaneet käyttää näitä rajapintoja toteutuksissaan. Erityisesti lokituksen käyttö on lisääntynyt sen ollessa kapseloituna roolirajapinnan avulla. Teorian kohdassa 3.3 on kuvattu ohjelmistokehysten käytön parantavan ohjelmistojen ylläpitoa lisäämällä modulaarisuutta. Esitetyt havainnot komponenttikehyksen vaikutuksista ovat linjassa tämän teorian kanssa.

Komponenttikehyksen, ja varsinkin komponenttikehyskontekstin suunnittelussa yksi kantava periaate on ollut puolustautuminen sen käyttäjien tekemiä virheitä vastaan. Tällä tavoitellaan pienempää vahingossa tehtävien virheiden määrää. Yksi konkreettinen esimerkki puolustautumisesta on poikkeusten heittäminen tunnistettavissa olevista ajonaikaista virhekonfiguraatioista. Tällöin komponenttikehyksen käyttäjän pitää tehdä tietoinen valinta poikkeuksen huomiotta jättämisestä, jos hän haluaisi jatkaa kirjoittamansa koodin suoritusta, ja tällaista valintaa ei tehdä vahingossa. Tämä näkyy testauksessa esille tulevien konfiguraatiovirheiden pienempänä määränä.

Asiakaskohtaisen konfiguraation (kohta 5.2) käytännössä ainoa erikoistamismekanismi on komponenttikehyksen kantaluokkien rakentajien paremetrien käyttö. Näiden

parametrien avulla komponenttikehykselle saadaan välitettyä sen vaatimat asiakaskohtaiset erikoistamisrajapintojen toteutukset. Räätälöinti erikoistamisrajapintojen kautta on tyypillistä koottaville kehyksille [24].

Poikkeuksen erikoistamisrajapintojen käyttöön tekee virheenkäsittelyn oletustoteutuksen erikoistaminen. Tässä oletustoteutuksen erikoistamismekanismina on periytyminen. Tämä erikoistamismekanismi on nähtävissä liitteessä C. Periyttämällä erikoistettu virheenkäsittelyn toteutus injektoidaan lopulta kuitenkin komponenttikehykselle rakentajan parametrina. Koska virheenkäsittely on komponenttikehyksen vaatima erikoistus, ja tässä erikoistetaan yhtä erikoistuksen toteutusta, voidaan katsoa, ettei tämä vaikuta tässä diplomityössä rakennetun komponenttikehyksen erikoistamirajapintojen tarkasteluun.

Koottaville kehyksille on tyypillistä niiden helppo käyttöönotto [12, s. 35]. Luvussa 6 kuvattujen jakelun ja konfiguraation hallinnan testaus- ja iteraatiokierrosten jälkeen käyttöönoton helppous on pitänyt paikkansa myös tässä diplomityössä toteutetun komponenttikehyksen osalta. Komponenttikehystä käyttämällä kehittäjä pääsee nopeimmillaan Dynamics CRM:n liitännäisten ja räätälöityjen työnkulkujen kehitystyössä alkuun reilusti alle viidessä minuutissa.

Komponenttikehyksen erikoistamiseen käytetyt rajapinnat ovat olleet vakaat komponenttikehyksen varhaisista testiversioista lähtien. Vakaat rajapinnat ovat yksi ohjelmistokehyksen tyypillinen tunnusmerkki [12]. Huomioiden muun muuassa komponenttikehyksen erikoistamismekanismit sekä käyttöönoton helppous, komponenttikehyksen voi katsoa olevan lähimpänä koottavaa kehystä.

Komponenttikehykselle oli kohdassa 4.4 esitetty ei-toiminnallinen vaatimus kehitystyön samanlaisuudesta ohjelmistokehittäjän näkökulmasta katsottuna. Tämä vaatimus koski sekä liitännäisten että räätälöityjen työnkulun vaiheiden kehitystyötä. Kuvasta 5.5 nähdään komponenttikehyksen ja sen avulla erikoistettavien luokkien periytymishierarkia ylätasolla kuvattuna. Tehdyn toteutuksen mukainen periytymishierarkia on hyvin samanlainen liitännäisten ja räätälöityjen työnkulun vaiheiden kesken. Kehittäjän näkökulmasta näiden kahden periytymishierarkian ainoat erot ovat kantaluokkien nimet sekä *Run()*-metodin saaman komponenttikehyskontekstin tarkka tyyppi. Nämä periytymishierarkiat vastaavat myös niistä ylätasolla esitettyä hierarkiaa. Komponenttikehyksen kantaluokkien tehty toteutus on nähtävissä liitteessä B. Komponenttikehyksen voidaan todeta vastaavan esitettyyn vaatimukseen kehitystyön samanlaisuudesta liitännäisten ja räätälöityjen työnkulun vaiheiden osalta.

8.3 Vastaavuus liiketoiminnan tarpeisiin

Komponenttikehyksen toteutus hyödyntää kohdan 8.1 mukaisesti useita ylläpidettävyyden parantamiseen tähtääviä tekniikoita. Kohdassa 2.2 kuvattiin komponenttikehykselle asetettuja liiketoiminnan hiljaisia tavoitteita. Niissä toivottiin jonkinlaisen järjestyksen saamista sellaisiin asioihin, joista monet pohjimmiltaan leikkaavat läpi koko järjestelmäkehityksen. Näitä järjestelmäkehityksen läpileikkaavia aspekteja on tarkasteltu teoriatasolla kohdassa 3.2. Asetettujen hiljaisten tavoitteiden tärkeimpinä läpileikkaavina asioina olivat virhetilanteiden yhtenäinen ajonaikainen hallinta sekä jäljittäminen jälkikäteen lokeista. Yleisemmin myös kehittäjittäin vaihtelevat kehityskäytännöt koettiin tietyssä mielessä läpi koko järjestelmäkehityksen ilmenevänä ongelmana.

Monet komponenttikehykselle asetetuista liiketoiminnan tavoitteista liittyvät toimitettavien järjestelmien laadullisiin asioihin. Kohdassa 3.2 kuvatun mukaisesti monien ei-toiminnallisten vaatimusten taustalla on pyrkimys saavuttaa laadullisia tavoitteita. Tarkastelen vastaavuutta liiketoiminnan tavoitteisiin tämän takia ensisijaisesti ei-toiminnallisten vaatimusten kautta. Komponenttikehyksen käyttö on nykyisellään uusissa Accountor Enterprisen Oy:n Dynamics CRM -projekteissa ennemmin vaatimus järjestelmän ylläpidettävyyden näkökulmasta kuin vapaaehtoinen valinta. Komponenttikehyksen kautta haetaan tasalaatuisuutta kohdassa 2.2 kuvattuihin ylläpidollisiin haasteisiin. Varsinkin komponenttikehyksen kautta tapahtuva yhtenäinen lokiin kirjoitus auttaa paljon ylläpidossa, kun kaikkia poikkeustilanteita voi aina jäljittää lokin kautta. Komponenttikehys on myös yhtenäistänyt kehityskäytäntöjä varsinkin luvussa 5 kuvattujen kantaluokkien ja komponenttikehyskontekstin kautta tarjottavien valmiiden toteutusten myötä. Tämä yhtenäistäminen on tarjonnut jonkin verran myös kehitystyön nopeutumista. Kehityskäytäntöjen yhtenäistyminen sekä työn nopeutuminen olivat molemmat kohdassa 2.2 mainittuja liiketoiminnan hiljaisia tavoitteita. Edellä mainittujen asioiden perusteella komponenttikehyksen käytön voidaan katsoa olevan liiketoiminnan näkökulmasta ensisijaisesti laadullinen valinta.

Ylläpidettävyyden näkökulmasta kohdan 4.4 ei-toiminnallisista vaatimuksista lokituksen käsittely ja virhetilanteiden hallinta ovat kohdassa 2.2 sivuttuja liiketoiminnan hiljaisia tavoitteita. Lokitukseen liittyvät ei-toiminnalliset vaatimukset on toteutettu joko kirjastorakenteen (kohta 5.1) tai komponenttikehyskontekstin (kohta 5.4) yhteydessä käsitellyissä toteutuksissa. Kaikki lokituksen hallinnan ei-toiminnalliset vaatimukset on saatu toteutettua. Myös kaikki lokitukseen liittyvät, kohdassa 4.3 kuvatut toiminnalliset vaatimukset saatiin toteutettua. Yrityksen liiketoiminnan näkökulmasta lokituksen toiminnalliset vaatimukset liittyvät valtaosin järjestelmän yl-

läpidettävyyden parantamiseen ja pienemmissä määrin kehityskäytäntöjen yhtenäistämiseen.

Virhetilanteiden hallintaan liittyvät ei-toiminnalliset vaatimukset on toteutettu kohdassa 5.3 selitetyn asiakaskohtaisen logiikan kutsun yhteyteen. Asiakaskohtaisen logiikan kutsu on ympäröity koodilohkolla, joka takaa kohdassa 5.2 kuvatun virheenkäsittelyn kutsumisen aina, mikäli asiakaskohtaisesta logiikasta kuplii käsittelemätön poikkeus (katso sivu 40). Tässä samaisessa poikkeuksen kiinni ottavassa lohkossa taataan myös kohdassa 4.3 esitettyjä lokituksen toiminnallisia vaatimuksia, kuten jokaisen liitännäisen suorituksen aloitus ja lopetus. Virhetilanteiden hallinnalle esitetyistä toiminnallisista vaatimuksista ei ole toteutettu esimerkkitoteutusta sille, miten poikkeuksien virheilmoituksista näytetään loppukäyttäjän kieliasetusten perusteella käännetyt versiot. Tämän vaatimuksen on todettu olevan toteutettavissa rakennetuilla virheenkäsittelyn toteutus- ja erikoistamismekanismeilla. Muut virhetilanteiden hallinnan toiminnalliset vaatimukset saatiin toteutettua.

Komponenttikehykseltä vaadittiin sen käyttöönoton olevan helpompaa ja toiminnallisuuden houkuttelevampaa kuin kehittäjän omien kirjastojen kopiointi (kohta 4.4). Tässä onnistuttiin, vaikkakin tähän vaatimukseen vastaaminen oli odotettua työläämpää. Käyttöönoton yhteydessä esiintyi kohdassa 7.3 kuvattua muutosvastarintaa huomattavasti ennakoitua voimakkaampana. Monet luvussa 6 kuvatut asiat ovat auttaneet käyttöönottokynnyksen madaltamisessa. Näitä asioita ovat esimerkiksi jakelukanavaksi valittu sisäinen NuGet-palvelu ja kääntyvä esimerkkitoteutus kohdan 5.2 mukaisesta asiakaskohtaisen konfiguraation erikoistamisesta. Myös kirjallinen ohje käyttöönoton vaiheista on auttanut käyttöönotossa. Ohje tarjoaa helppoa ja varmaa toistettavuutta suhteellisen harvoin tehtävään komponenttikehyksen käyttöönottoon.

Käyttöönoton houkuttelevuuden ei-toiminnalliseen vaatimukseen liittyy useiden kohdan 4.3 toiminnallisten vaatimusten toteuttaminen. Minimivaatimusten voidaan katsoa täyttyneen, kun komponenttikehyskonteksti (kohta 5.4) tarjoaa Dynamics CRM:n tapahtumakäsittelylinjaston suorituskontekstiin liittyviä asioita kuten *IOrganizationService*[1]. Komponenttikehyksellä on toiminnallinen vaatimus tiettyjen yleisten toimenpiteiden tekemisen helpottamisesta. Tämän vaatimuksen täyttämistä varten tehdyt toteutukset ovat osaltaan myös käyttöönoton houkuttelevuutta lisääviä asioita. Yleisesti tehtävät toiminnot liittyvät monessa tapauksessa Dynamics CRM:n suorituskontekstin kanssa toimimiseen.

[1]Katso *IOrganizationService* kohdan 3.4 sivulta 26.

Yksi tällainen kehittäjien yleisesti tekemä toimenpide Dynamics CRM:n suorituskontekstin kanssa on *PreImage*- ja *PostImage*-vedosten[2] nouto. Kohdassa 5.4 mainittiin tietyn käytännön mukainen pakotus näiden vedosten nimeämisessä. Tähän on kaksi syytä. Toinen näistä on halu yhtenäistää kehityskäytäntöjä niin, että nimeämiset ovat yhtenevät asiakasprojektista ja kehittäjästä riippumatta. Toinen syy liittyy kohdassa 5.4 kuvattuun tapaan, jolla Dynamics CRM -alusta välittää vedokset ohjelmistokehittäjän käyttöön: vedokset yksilöidään nimellä, joten samannimisistä vedoksista vain viimeisin jää voimaan. Tällöin kehittäjä ei välttämättä saa vedoksesta kaikkia olettamiaan tietoja. Tämä voi johtaa hyvin hankalasti jäljitettäviin virheisiin. Edellä oleva kuvaus on esimerkki siitä, miten komponenttikehyksen avulla pyritään saavuttamaan liiketoiminnan laadullisia tavoitteita ehkäisemällä virheiden syntymistä. Laadullisiin tavoitteisiin pyritään ohjaamalla toteutus komponenttikehyksen rajapintasuunnittelun avulla poispäin virheherkistä ratkaisuista samalla yhtenäistäen sitä, miten eri kehittäjät kehittävät järjestelmää. Lisäksi edellä kuvattu on esimerkki siitä, miten komponenttikehyksen käyttöönotosta pyritään tekemään kehittäjille houkutteleva auttamalla heitä heidän yleisesti tekemissä operaatioissaan.

Komponenttikehyksen ei-toiminnallisena vaatimuksena oli tunnottomuus siihen, millaista asiakaslogiikkaa sen päälle toteutetaan. Ainoa komponenttikehyksen tekemä oletus kohdassa 5.3 kuvatusta asiakaslogiikasta on se, että se on olemassa. Tunnottomuusvaatimuksen voi katsoa toteutuneen. Kehittäjä voi siis käyttää komponenttikehystä kaikkeen, mikä on mahdollista Dynamics CRM -alustan asettamissa rajoissa. Tähän liittyy toiminnallinen vaatimus asiakaslogiikan kehitystyössä tarvittavien kantaluokkien tarjoamisesta. Komponenttikehys toteuttaa nämä kantaluokat kuten luvussa 5 on esitetty. Asiakaskohtaisen logiikan toteuttamisen kannalta kohdassa 5.3 kuvattu komponenttikehyksen *Run()*-metodi on Dynamics CRM -kehittäjän näkökulmasta tutun oloinen. Nyt kun *Run()*-metodia tarkastellaan jälkeenpäin, se on idealtaan käytännössä jatkojalostettu versio siitä, miten asiakaskohtainen logiikka tehdään Dynamics CRM:ään ilman komponenttikehystä[3].

Koska komponenttikehys voi olla käytössä jokaisessa tai ainakin hyvin monessa asiakasjärjestelmässä, sen oikea toiminta pitää varmistaa. Kohdassa 4.4 esitettiin ylläpidettävyyteen liittyvä ei-toiminnallinen vaatimus siitä, ettei mikään komponenttikehyksen päälle rakennettu järjestelmä ala toimia väärin komponenttikehyksen muutoksen tai uuden ominaisuuden takia. Tähän vaatimukseen panostettiin huomattavan paljon muun muuassa yksikkötestauksen muodossa. Lopputuloksena komponenttikehyksen julkinen rajapinta saatiin kehitysprosessin aikana (kohta 7.1) käytännössä kokonaan automaattisten yksikkötestien piiriin. Tällä testausmekanismilla on

[2]Katso *PreImage*- ja *PostImage*-vedokset kohdan 3.4 sivulta 18.
[3]Katso asiakaskohtaisen logiikan toteutus *Execute()*-metodilla kohdan 3.4 sivulta 25.

jo onnistuttu huomaamaan muutostilanteissa tehtyjä ohjelmointivirheitä. Liiketoiminnan näkökulmasta tämä testauskattavuus on merkittävää. Testauskattavuuden takia komponenttikehystä voidaan varauksetta suositella kaikkiin asiakasprojekteihin.

Komponenttikehys on otettu käyttöön melko laajasti. Selvällä enemmistöllä Accountor Enterprise Solutions Oy:n Dynamics CRM -ohjelmistokehittäjistä on nykyään jollain tasolla kokemusta komponenttikehyksen käyttämisestä. Käyttäjiltä on tullut noin kolme vuotta sitten olleen virallisen julkaisun jälkeen joitakin lisäominaisuustoivomuksia. Näitä ominaisuuksia on lisätty komponenttikehykseen. Komponenttikehyksen käyttöönotolla on jo nähtävissä jonkin verran hyötyä kehitysmallin yhtenäistymisellä infrastruktuuritasolla. Tämä yhtenäistyminen tekee toteutuksesta tutumman oloista jos ja kun kehittäjiä pitää vaihtaa asiakasprojekteista toiseen. Ohjelmointimallin yhtenäistymisen ansiosta kehittäjien ei enää nykyisin tarvitse tutustua uuden asiakasprojektin teknisiin nyansseihin, kuten lokitukseen ja parametriston hallintaan. Teknisten arkkitehtien pitää toki edelleen hallita projektien teknisetkin nyanssit. Komponenttikehyksen tuomasta kehityskäytäntöjen yhtenäistymisestä huolimatta kohdassa 2.2 mainitut ohjelmistokehittäjien yksilölliset kehityskäytännöt ovat edelleen jossain määrin kehitystyön haasteena.

Valtaosa komponenttikehyksen käyttäjistä ei ole kommentoinut komponenttikehyksen toimintoja ja rakennetta sen enempää positiivisesti kuin negatiivisestikaan. Käyttöönoton helppoudesta ja nopeudesta on sen sijaan on kuulunut useilta kehittäjiltä positiivista ja yllättynyttä palautetta. Kun Visual Studio -kehitysympäristössä aloittaa luomaan uutta projektia Dynamics CRM -kehitykseen, aikaa kuluu alle viisi minuuttia siihen, kun komponenttikehys on otettu käyttöön ja lähdekoodi kääntyy. Kehitystyön aloitukseen kuluvaa asetusaikaa ilman komponenttikehyksen käyttöä ei ole mitattu systemaattisesti. Kehitystyön aloitus ilman komponenttikehystä kesti diplomityön kirjoittajan kokemusten mukaan noin yhdestä kolmeen tuntia. Tähän vaikutti se, miten laajasti toteutuksia kopioitiin aiemmista asiakastoteutuksista. Komponenttikehyksen käyttö on siis nopeuttanut kehitystyön aloitusta huomattavasti verrattuna ohjelmistokehittäjien aiemmin käyttämiin tapoihin.

Komponenttikehyksen voidaan ajatella Ossherin [102] ja Tarrin [112] perusteella muodostavan tietystä näkökulmasta tarkasteltuna itsenäisen moduulin järjestelmän infrastruktuuritason läpileikkaavaan ongelmaan. Myös kohdan 3.2 voidaan tulkita tukevan tätä ajatusta, ja sieltä erityisesti seuraava Bergmansin [4] toteamus: jos järjestelmän läpileikkaavaa ongelmaa ei saada kuvattua yhtenä moduulina, ratkaisun hyödynnettävyys ja uudelleenkäytettävyys mitä todennäköisimmin kärsivät. Tällainen infrastruktuurinäkökulman erottaminen todennäköisesti jättää asiakasprojek-

teissa enemmän aikaa ja mahdollisuuksia keskittyä asiakkaan liiketoiminnasta kumpuavien tarpeiden ratkaisemiseen. Yksittäisen asiakkaalle tehtävän liitännäisen tai räätälöidyn työnkulun vaiheen toteutustyön kesto on kohdan 2.1 mukaisesti yleensä korkeintaan muutamia henkilötyöpäiviä. Tätä työmäärää voidaan peilata komponenttikehyksen tarjoamaan kehitystyön aloituksen nopeutumiseen sekä infrastruktuuriaspektin ja asiakkaan liiketoiminnan tarpeiden erottamisen hyötyihin. Komponenttikehyksen voikin todeta antavan suhteellisen paljon lisäaikaa varsinaiseen asiakkaan ongelman ratkaisemiseen erityisesti pienissä toteutuksissa.

8.4 Tarkastelu muiden asioiden suhteen

Kaikki komponenttikehykselle esitetyt vaatimukset eivät ole liiketoiminnan tarpeista tulevia vaatimuksia. Komponenttikehyksen suunnittelussa on pyritty ohjaamaan yrityksen kehittäjien kehitystapoja hyväksi havaittujen käytäntöjen suuntaan. Tällaisia käytäntöjä ovat esimerkiksi asetustietueiden käyttäminen ympäristökohtaiseen parametrisointiin (kohta 5.2) ja staattisesti sidotun ohjelmointitavan käyttö. Tämä ohjelmointitapa on nimenomaisesti Microsoftin suosittelema tapa Dynamics CRM -kehitystyöhön [41]. Tämän ohjelmointitavan laajemman käytön on todettu olevan yritystasolla hyvä asia. Jaeltava NuGet-paketti sisältääkin erikoistamisen esimerkin ainoastaan staattisesti sidotun ohjelmointitavan käytöstä. Saman perustelun takia erikoistamisen esimerkissä on valmiina myös asetustietueiden ja lokikirjaston käyttö. Tämän diplomityön tulosten tarkastelun näkökulmasta nämä asiat ovat paitsi teknisiä ja tarjoavat tiettyjä toiminnallisuuksia, mutta ne myös sivuavat kohdassa 2.2 mainittuja liiketoiminnan tavoitteita ylläpidollisten ongelmien vähentämisestä.

Joskus on tarvetta jäljittää asiakkaiden Dynamics CRM -ympäristöissä, miksi jokin loppukäyttäjän tekemä toiminto toimii hitaasti. Jos tällaiseen toimintoon liittyy liitännäisiä tai räätälöityjä työnkulun vaiheita, tutkinnassa voidaan käyttää apuna komponenttikehyksen profilointiominaisuutta. Tällä kohdassa 4.3 esitetyllä toiminnallisella vaatimuksella saadaan kerättyä tietoa, joka auttaa rajaamaan sitä, onko hitaan suorituksen syy mahdollisesti jossakin liitännäisessä tai räätälöidyssä työnkulun vaiheessa. Komponenttikehyksen profilointiominaisuudet ovat helpottaneet tuotantoympäristöissä hitaasti toimivien komponenttien jäljitystä. Profilointitiedot on saatu kerättyä tällaisissa ympäristöissä tekemällä yksinkertainen konfiguraatiomuutos.

Komponenttikehyksen idean syntyaikoihin oli tunnistettu entistä suuremman asiakasjoukon valitsevan tulevaisuudessa Dynamics CRM -järjestelmän pilviversion paikallisille palvelimille asentamisen asemasta. Näistä pilviversio asettaa kohtuullisen

tiukat reunaehdot sille, mitä siellä suoritettava ohjelmakoodi saa tehdä. Näiden asioiden takia liiketoiminnan suunnasta tuli ei-toiminnallinen vaatimus komponenttikehyksen toimivuudesta sekä pilvessä että paikallisesti asennettuna. Tämä vaatimus on esitetty kohdassa 4.4. Tähän vaatimukseen vastaaminen osoittautui yllättävän helpoksi. Ainoa kohta, jossa samaa toimintoa ei saatu toimimaan täysin identtisesti myös pilviversiossa, liittyi tiettyyn profiloinnissa kerättävään tietoon. Tätä kyseistä tietoa ei yritetä kerätä, kun profiloidaan pilviympäristöä, joten tällä eroavaisuudella ei ole vaikutusta komponenttikehyksen käyttöön.

Komponenttikehyksen rakentamisen epäsuoria hyötyjä ovat olleet erilaisten ohjelmistokehityksen käytäntöjen tuleminen kehittäjien tietoisuuteen. Varsinkin asioiden abstrahoinnin ja toisistaan erottamisen tärkeys järjestelmäkehityksen yhteydessä on lisääntynyt. Konkreettisesti tämä on näkynyt esimerkiksi toteutettujen liitännäisten vastuiden tarkempana erotteluna ja sitä kautta liitännäisen keskimääräisen koon pienentymisenä. Toinen konkreettinen, mutta epäsuora hyöty on ollut kohdassa 6.2 mainitun sisäisen NuGet-palvelun käyttöönoton jälkeen yleistynyt ulkoisista NuGet-palveluista löytyvien valmiskirjastojen käyttö. Osa tarvittavasta toteutuksesta on siis erotettu ulkoisen NuGet-kirjaston tarjoaman rajapinnan[4] avulla toteutetuksi. Valmisratkaisujen lisääntyvä käyttö on osaltaan lisännyt asiakkaan ongelmien ratkaisemiseksi käytettävissä olevaa aikaan.

[4]Tässä yhteydessä rajapinnaksi käsitetään ulkoisen kirjaston koko julkinen toteutus.

9. JOHTOPÄÄTÖKSET

Tietojärjestelmien koko historian ajan on pyritty löytämään niiden kehitystyötä helpottavia ratkaisuja. Tässä diplomityössä rakennettu komponenttikehys ei poikkea tästä linjasta. Komponenttikehys tehtiin auttamaan yritystä Microsoftin Dynamics CRM -alustan päälle rakennettavien järjestelmien kehitystyössä. Komponenttikehykselle oli asetettu tiettyjä yrityksen liiketoiminnasta kumpuavia ja osin melko epämääräisiäkin tavoitteita ja odotuksia. Näitä olivat esimerkiksi kehitystyön nopeutus, käytäntöjen yhtenäistäminen ja jonkin järjestyksen saaminen kehitystyössä vallitsevaan anarkiaan.

Näiden epämääräisten tavoitteiden lisäksi komponenttikehykselle kerättiin joukko konkreettisemmin ilmaistuja vaatimuksia. Näitä kerättiin komponenttikehyksen tulevilta käyttäjiltä eli ohjelmistokehittäjiltä sekä teknisiltä arkkitehdeiltä. Liiketoiminnan vaatimuksia konkretisoitiin esimieheni kanssa käydyissä keskusteluissa.

Komponenttikehys rajattiin tukemaan Dynamics CRM -alustaan tehtävien liitännäisten ja räätälöityjen työnkulun vaiheiden toteutusta. Toteutus tehtiin C#-kielellä. Dynamics CRM -alustan tapahtumapohjainen liitännäisarkkitehtuuri ohjasi komponenttikehyksen suunnittelua hyvin voimakkaasti. Komponenttikehys on sitä käyttävän kehittäjän näkökulmasta hyvin tutun oloinen verrattaessa tilannetta kehitystyöhön ilman komponenttikehystä. Komponenttikehys tarjoaa järjestelmäkehitykseen kuitenkin muutamia merkittäviä etuja. Nämä edut muodostuvat kehittäjille tarjottujen roolirajapintojen ja niiden takana olevien puolustautuvien toteutusten kautta.

Ohjelmistokehittäjät tekevät säännöllisesti virheitä toimiessaan Dynamics CRM -alustan kanssa. Komponenttikehyksen yksi kantava suunnitteluperiaate on ollut puolustautuminen näiltä virheiltä. Tämä näkyy monissa pienissä rajapintojen yksityiskohdissa. Dynamics CRM ei esimerkiksi tarjoa vakio-ominaisuutena mahdollisuutta tunnistaa kehittäjän liitännäisen toteutuksessa tekemiä virheoletuksia käytettävissä olevasta konfiguraatiosta. Komponenttikehykseen puolestaan on rakennettu tarkistusmekanismeja, joissa heitetään poikkeus, mikäli ajonaikaisessa Dynamics CRM -ympäristössä oleva konfiguraatio poikkeaa kehittäjän olettamasta konfiguraatiosta.

Toisenlainen esimerkki puolustautumisesta on helpottaa hyväksi havaittujen käytäntöjen hyödyntämistä tarjoamalla näitä käytäntöjä tukevia metodeja ja rajapintoja. Jälkimmäisessä esimerkissä ei puolustauduta niinkään virheitä vastaan, vaan toimitaan ylläpidettävyyden parantamisen puolesta.

Komponenttikehyksen voi mieltää olevan poikkileikkaava infrastuktuurikerros Dynamics CRM -alustan ja asiakkaalle toteutetun sovelluslogiikan välillä. Monet tavoitteiksi asetetut toiminnalliset ja ei-toiminnalliset vaatimukset liittyivät infrastruktuuritason kanssa toimimiseen. Näihin vaatimuksiin vastaaminen oli loppujen lopuksi yllättävän helppoa komponenttikehyksen arkkitehtuurin ja toteutuksen näkökulmasta. Käytännössä kaikki arkkitehtuuriin liittyvät vaatimukset saatiin toteutettua. Ainoa puuttuva vaatimuksen toteutus on lokalisoitujen virheilmoitusten tukeminen. Tämänkin vaatimuksen osalta tiedetään, miten se pitää toteuttaa, kun tarve tulee vastaan asiakasprojektissa.

Tämän diplomityön tekoon on kulunut melko pitkä ajanjakso. Komponenttikehyksen käyttöönotosta yrityksen sisällä on kulunut noin kolme vuotta. Tämä ajanjakso on tuonut mukanaan hyvän tarkasteluperspektiivin komponenenttikehykselle asetettujen liiketoiminnan tavoitteiden täyttymiseen. Komponenttikehyksen käyttö on yrityksessä jo laajalle levinnyttä. Yrityksen asettamiin tavoitteisiin kehitystyön nopeutumisesta ja kehityskäytäntöjen yhtenäistymisestä on onnistuttu vastaamaan. Kehityskäytäntöjen yhtenäistymisen hyödyt ovat alkaneet näkyä asiakasjärjestelmien laadussa. Hyötyjä näkyy erityisesti muutostilanteissa, kun järjestelmän parissa työskentelevät henkilöt vaihtuvat. Lisäksi komponenttikehyksen käyttöönoton nopeus on todella merkittävä hyöty pienissä toteutustarpeissa. Komponenttikehys pienentää kehitystyön alussa tapahtuvaa asetusaikaa muutamista tunneista minuutteihin tarjoten samalla myös muut tässä luvussa mainitut hyödyt.

Liiketoiminnan tavoitteiden täyttyminen on tapahtunut kuitenkin huomattavasti odotettua hitaammin. Tähän on vaikuttanut käyttöönottovaiheessa kehittäjien suunnasta koettu muutosvastarinta. Muutosvastarinnan takia komponenttikehys on levinnyt yrityksessä käyttöön melko hitaasti. Muutosvastarinta oli voimakasta varsinkin ensimmäisessä isommassa asiakasprojektiryhmässä, jossa pilotoitiin itsenäistä käyttöönottoa. Isoksi muutosvastarinnan syyksi koettiin pilotoinnin aikaan puuttunut käyttöönottoa tukeva dokumentaatio sekä pilottiasennuksen jälkeen kääntymätön esimerkkikonfiguraation lähdekoodi. Tämä johti vastareaktiona kilpailevan ratkaisun kehitystyön aloittamiseen. Kilpailevan ratkaisun käytön leviäminen saatiin rajattua, sillä yrityksessä ei haluttu samaan tarpeeseen kahta erillistä ratkaisua. Kilpaileva ratkaisu on kuitenkin edelleen käytössä muutamassa isossa projektissa. NuGet-jakelun ajateltiin vähentävän kilpailevien kehityshaarojen syntymistä. Tällä

tarkoitettiin rakennetun komponenttikehyksen kilpailevia haaroja, joita ei syntynyt. Täysin uuden kilpailevan ratkaisun mahdollisuutta ei osattu ennakoida. Komponenttikehyksestä saatavan hyödyn voi ennakoida kasvavan tulevaisuudessa johtuen sen käytön jatkuvasta lisääntymisestä ja muutosvastarinnan vähentymisestä.

Yksi kimmoke kilpailevalle ratkaisulle saattoi olla komponenttikehyksen käyttöönoton mukanaan tuoma yhtenäinen ja erilaiseksi koettu käytäntö. Nämä asiat koettiin mahdollisesti uhaksi ohjelmistokehittäjän taiteelliselle vapaudelle. Kuten Knuth [26] on esittänyt, ohjelmistokehittäjiä voi pitää taiteilijoina, joista jokainen pitää eniten omasta ohjelmistokehitystyylistään. Muutosvastarinnan hallinta on erittäin tärkeää sisäisten kehitystyökalujen kehityshankkeiden yhteydessä, jotta uuden kehitystyökalun käyttöönotto onnistuu ilman nyt havaitun kaltaista vastustusta. Mielestäni tässä auttaa aktiivinen kommunikaatio tulevista ominaisuuksista. Lisäksi suosittelen käyttöönottoa tukevan dokumentaation ja sujuvan käyttöönottoprosessin hiomista käyttöönoton pitkittymisen kustannuksellakin. Dokumentaation merkitystä ohjelmistokehyksen onnistumiselle on tuotu esille esimerkiksi Faydin [12] toimesta.

Diplomityössä tehdyn komponenttikehyksen rakentamisen voi katsoa olevan *Design science* -ajattelun [21] mukainen: komponenttikehys rakennettiin täyttämään tiettyä Accountor Enterprise Solutions Oy:n liiketoiminnan kautta kummunnutta tarvetta. Kohdan 2.2 mukaisesti komponenttikehyksen kehitystyöllä oli tavoitteina nopeuttaa tiettyjen Dynamics CRM -kehitystöiden toteutusta sekä hieman pidemmällä aikavälillä yhtenäistää yrityksen Dynamics CRM -kehityskäytäntöjä. Vertailtaessa tehtyä toteutusta luvussa 2 esitettyihin toteutuksen lähtökohtiin ja tavoitteisiin, näistä monien huomataan toteutuneen tai parantuneen. Osa muutoksista on suoraa seurausta komponenttikehyksen kehityksestä, kun taas osa muutoksista johtuu epäsuorista vaikutuksista.

Komponenttikehyksen mahdollisesti merkittävin epäsuora vaikutus on ollut yrityksen teknisten arkkitehtien ja kehittäjien keskuudessa lisääntynyt ymmärrys asioiden abstrahoinnin ja toisistaan erottamisen tärkeydestä. Komponenttikehys on auttanut ymmärryksen lisääntymisessä nostamalla näitä asioita keskusteluun yrityksen sisällä, jolloin niihin on alettu kiinnittää enemmän huomiota. Tällaiset vaikutukset ovat hankalasti mitattavia varsinkin, kun vaikutukset huomattiin vasta jälkikäteen. Saamani vaikutelman mukaan tämä tietoisuuden lisääntyminen on ollut enimmäkseen tiedostamatonta. Tämä on välillisesti auttanut komponenttikehyksen kehittämiselle asetettujen tavoitteiden saavuttamisessa. Komponenttikehyksen suoria hyötyjä laaja-alaisempia hyötyjä voi ennakoida epäsuorien vaikutusten kautta. Nämä vaikutukset ohjaavat yrityksen yleistä kulttuuria ja toimintatapaa kestävämpien ohjelmistokehityksen käytäntöjen suuntaan. Koen tämänkaltaisten pehmeiden ja epä-

suorien vaikutusten kautta saatavien hyötyjen selvittämisen olevan varteenotettava tutkimuskysymys jossakin yrityksessä tehtävän *Design science* -tyylisen kehitystyön yhteydessä.

Saatujen kokemusten valossa komponenttikehyksen tyylistä ratkaisua voi suositella yhtenäistämään käytäntöjä, parantamaan järjestelmän oikeaoppista kehitystyötä ja nopeuttamaan kehitystyötä ainakin Dynamics CRM -järjestelmissä. Yrityksessä on toteutettu kirjasto hyvin samanlaisella idealla auttamaan selaimessa ajettavien Dynamics CRM:n JavaScript-toteutuksien kehitystyössä. Tämä kirjasto tarjoaa sekä kehittäjien että liiketoiminnan näkökulmasta samoja hyötyjä kuin diplomityössä toteutettu komponenttikehys. Näiden kokemusten perusteella suositusta komponenttikehyksen tyylisen ratkaisun käytöstä voi luultavasti laajentaa muihinkin saman tyylisiin järjestelmiin. Esimerkkejä vastaavista järjestelmistä ovat Oracle Netsuite [101] ja SalesForce[107]. Näistä kummassakin on melko samantyylinen tapahtumapohjainen ja kontekstiin sidottu ohjelmakoodilla tehtävä laajentamismekanismi [110] kuin Dynamics CRM -alustassakin.

LÄHTEET

[1] David Ameller. "'Non-Functional Requirements as drivers of Software Architecture Design'". Tohtorinväitöskirja. Departament de Llenguatges i Sistemes Informàtics, 2014, s. 244. URL: http://www.tdx.cat/handle/10803/144942.

[2] Malcolm Atkinson & Ronald Morrison. Data Types and Persistence. In: Malcolm Atkinson, Peter Buneman & Ronald Morrison (ed.) Types, Bindings and Parameters in a Persistent Environment, Springer, 1988, Berlin, s. 3–20.

[3] Andy Baron. Inheritance and Interfaces. Technical article. Microsoft, helmikuuta 2002. URL: https://msdn.microsoft.com/en-us/library/ms973861.aspx (viitattu 05. 05. 2016).

[4] Lodewijk Bergmans & Mehmet Aksit. Composing Crosscutting Concerns Using Composition Filters. Communications of the ACM 44(2001)10, pp. 51–57.

[5] Microsoft Dynamics CRM Team Blog. How to Reference Assemblies from Plug-ins -blogikirjoitus. [viitattu 24.04.2016]. Saatavissa https://blogs.msdn.microsoft.com/crm/2010/11/09/how-to-reference-assemblies-from-plug-ins/.

[6] Pierre Bourque & Richard E. Fairley, toim. Guide to the Software Engineering Body of Knowledge. Version 3.0. IEEE Computer Society, 2014.

[7] Don Box & Anders Hejlsberg. LINQ: .NET Language-Integrated Query. .NET Development article. Microsoft, helmikuuta 2007. URL: https://msdn.microsoft.com/en-us/library/bb308959.aspx (viitattu 12. 03. 2016).

[8] Lawrence Chung & Julio Cesar Sampaio do Prado Leite. On Non-Functional Requirements in Software Engineering. In: Alexander T. Borgida, Vinay K. Chaudhri, Paolo Giorgini & Eric S. Yu (ed.) Conceptual Modeling: Foundations and Applications: Essays in Honor of John Mylopoulos, Springer, 2009, Berlin, s. 363–379.

[9] Lawrence Chung, Sam Supakkul, Nary Subramanian, José Luis Garrido, Manuel Noguera, Maria V. Hurtado, María Luisa Rodríguez & Kawtar Benghazi. Relating Software Requirements and Architectures. In: Paris Avgeriou, John Grundy, G. Jon Hall, Patricia Lago & Ivan Mistrík (ed.) Springer, 2011, Berlin, pp. 91–109.

[10] Simon Cropp. Costure add-in for Fody. [viitattu 02.03.2016]. Saatavissa https://github.com/Fody/Costura.

[11] Dino Esposito. Cutting Edge - Pros and Cons of Data Transfer Objects. MSDN Magazine 24(2009)8.

[12] Mohamed Fayad & Douglas C. Schmidt. Object-oriented Application Frameworks. Communications of the ACM 40(1997)10, pp. 32–38.

[13] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. [viitattu 05.11.2017]. Saatavissa https://www.martinfowler.com/articles/injection.html.

[14] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Boston Addison-Wesley, 1995. 395 p.

[15] Punit Ganshani. 5 steps to targeting multiple .NET frameworks. [viitattu 29.04.2016]. Saatavissa http://blogs.msmvps.com/punitganshani/2015/06/21/5-steps-to-targeting-multiple-net-frameworks/.

[16] Roger Gilchrist. Scalable Customization Design in Microsoft Dynamics CRM. White paper. Microsoft, maaliskuuta 2015.

[17] Martin Glinz. On Non-Functional Requirements. Delhi, India 2007. pp. 21–26.

[18] Ian Goldberg, David Wagner, Randi Thomas & Eric A. Brewer. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. San Jose, California 1996, USENIX Association. pp. 1–1.

[19] Mark Grand. Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML. 2nd. New York John Wiley & Sons, Inc., 2002.

[20] Phil Haack. NuGet - Manage Project Libraries with NuGet. MSDN Magazine 26(2011)11.

[21] Alan R. Hevner, Salvatore T. March, Jinsoo Park & Sudha Ram. Design Science in Information Systems Research. MIS Quarterly 28(2004)1, s. 75–105.

[22] Christine Hofmeister, Philippe Kruchten, Robert L. Nord, Henk Obbink, Alexander Ran & Pierre America. A general model of software architecture design derived from five industrial approaches. Journal of Systems and Software 80(2007)1, pp. 106–126.

[23] Katrin Hussinger & Annelies Wastyn. In Search for the Not-Invented-Here Syndrome: The Role of Knowledge Sources and Firm Success. ZEW Discussion Paper 11-048. Mannheim: Centre for European Economic Research, 2011. URL: http://ftp.zew.de/pub/zew-docs/dp/dp11048.pdf (viitattu 10. 11. 2018).

[24] Ralph E. Johnson & Brian Foote. Designing Reusable Classes. Journal of Object-Oriented Programming 1(1988)2, pp. 22–35.

[25] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier & John Irwin. ECOOP'97 — Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings. Berlin 1997, Springer. pp. 220–242.

[26] Donald E. Knuth. Computer Programming as an Art. Communications of the ACM 17(1974)42, pp. 667–673.

[27] Kai Koskimies & Tommi Mikkonen. Ohjelmistoarkkitehtuurit. Helsinki Talentum, 2005. 250 s.

[28] Milosz Krajewski. LibZ. [viitattu 24.04.2016]. Saatavissa https://github.com/MiloszKrajewski/LibZ.

[29] Craig Larman. Protected Variation: The Importance of Being Closed. IEEE Software 18(2001)3, pp. 89–91.

[30] Barbara Liskov. Keynote Address - Data Abstraction and Hierarchy. SIGPLAN Notices 23(1987)5, pp. 17–34.

[31] Juval Lowy. An Introduction to C# Generics. Technical article. Microsoft, tammikuuta 2005. URL: https://msdn.microsoft.com/en-us/library/ms379564%28v=vs.80%29.aspx (viitattu 05. 05. 2016).

[32] Robert C. Martin. The Dependency Inversion Principle. The C++ Report 8(1996).

[33] Robert C. Martin. The Liskov Substitution Principle. The C++ Report 8(1996).

[34] Robert C. Martin. The Open-Closed Principle. The C++ Report 8(1996).

[35] Danijel Matić, Dino Butorac & Hrvoje Kegalj. Data access architecture in object oriented applications using design patterns. Dubrovnik, Croatia 2004, IEEE. pp. 595–598.

[36] Johannes Mayer, Ingo Melzer & Franz Schweiggert. Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays, NODe 2002 Erfurt, Germany, October 7–10, 2002 Revised Papers. In: Mehmet Aksit, Mira Mezini & Rainer Unland (ed.) Springer, 2003, Berlin, pp. 87–102.

[37] Bertrand Meyer. Object-oriented Software Construction. 2nd. Upper Saddle River Prentice-Hall, Inc., 1997. 1254 p.

[38] Microsoft. An introduction to NuGet. [viitattu 20.11.2018]. Saatavissa https://docs.microsoft.com/en-us/nuget/what-is-nuget.

[39] Microsoft. Asynchronous service in Microsoft Dynamics CRM 2013. [viitattu 17.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328021%28v=crm.6%29.aspx.

[40] Microsoft. Best practices for developing with Microsoft Dynamics CRM 2013. [viitattu 24.04.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg509027%28v=crm.6%29.aspx.

[41] Microsoft. Best practices for developing with Microsoft Dynamics CRM 2013. [viitattu 22.03.2015]. Saatavissa https://msdn.microsoft.com/en-us/library/gg509027%28v=crm.6%29.aspx.

[42] Microsoft. Build queries with LINQ (.NET language-integrated query) (Microsoft Dynamics CRM 2013). [viitattu 12.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328028%28v=crm.6%29.aspx.

[43] Microsoft. CodeActivity.Execute(CodeActivityContext) Method. [viitattu 05.11.2018]. Saatavissa https://docs.microsoft.com/en-us/dotnet/api/system.activities.codeactivity.execute.

[44] Microsoft. Create a custom workflow activity (Microsoft Dynamics CRM 2013). [viitattu 10.12.2017]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328515(v=crm.6).aspx.

[45] Microsoft. Create early bound entity classes with the code generation tool (CrmSvcUtil.exe) (Microsoft Dynamics CRM 2013). [viitattu 12.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg327844%28v=crm.6%29.aspx.

[46] Microsoft. Create your own actions (Microsoft Dynamics CRM 2013). [viitattu 02.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/dn481600%28v=crm.6%29.aspx.

[47] Microsoft. Creating Templates for Projects and Items in Visual Studio. [viitattu 29.04.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/ms247121%28v=vs.120%29.aspx.

[48] Microsoft. Custom workflow activities (workflow assemblies) (Microsoft Dynamics CRM 2013). [viitattu 02.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg309745%28v=crm.6%29.aspx.

[49] Microsoft. Debug a plug-In (Microsoft Dynamics CRM 2013). [viitattu 16.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328574%28v=crm.6%29.aspx.

[50] Microsoft. Early and Late Binding (Visual Basic Programming Guide). [viitattu 28.02.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/0tcf61s1%28v=vs.130%29.aspx.

[51] Microsoft. Enterprise Library (Microsoft patterns & practices). [viitattu 28.02.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/ff648951.aspx.

[52] Microsoft. Entity Class (Microsoft Dynamics CRM 2013). [viitattu 14.05.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/microsoft.xrm.sdk.entity%28v=crm.6%29.aspx.

[53] Microsoft. EntityImageCollection Class (Microsoft Dynamics CRM 2013). [viitattu 17.11.2018]. Saatavissa https://docs.microsoft.com/en-us/previous-versions/dynamicscrm-2013/crm.6/gg308458%28v%3dcrm.6%29.

[54] Microsoft. Event execution pipeline (Microsoft Dynamics CRM 2013). [viitattu 02.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg327941%28v=crm.6%29.aspx.

[55] Microsoft. Global Assembly Cache. [viitattu 24.04.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/yf1d93sz.aspx.

[56] Microsoft. Handle exceptions in plug-ins (Microsoft Dynamics CRM 2013). [viitattu 12.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg334685%28v=crm.6%29.aspx.

[57] Microsoft. IExecutionContext.CorrelationId Property (Microsoft Dynamics CRM 2013). [viitattu 24.04.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/microsoft.xrm.sdk.iexecutioncontext.correlationid%28v=crm.6%29.aspx.

[58] Microsoft. IExecutionContext.Depth Property (Microsoft Dynamics CRM 2013). [viitattu 02.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/microsoft.xrm.sdk.iexecutioncontext.depth%28v=crm.6%29.aspx.

[59] Microsoft. IExecutionContext.PreEntityImages Property (Microsoft Dynamics CRM 2013). [viitattu 17.11.2018]. Saatavissa https://docs.microsoft.com/en-us/previous-versions/dynamicscrm-2013/crm.6/gg326980(v=crm.6).

[60] Microsoft. Introduction to entities in Microsoft Dynamics CRM 2013. [viitattu 06.05.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg309396%28v=crm.6%29.aspx.

[61] Microsoft. Introduction to the event framework (Microsoft Dynamics CRM 2013). [viitattu 02.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg334361%28v=crm.6%29.aspx.

[62] Microsoft. IOrganizationService.Delete Method (Microsoft Dynamics CRM 2013). [viitattu 17.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg326162%28v=crm.6%29.

[63] Microsoft. IOrganizationService.Retrieve Method (Microsoft Dynamics CRM 2013). [viitattu 17.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg326164%28v=crm.6%29.aspx.

[64] Microsoft. IOrganizationService.RetrieveMultiple Method (Microsoft Dynamics CRM 2013). [viitattu 17.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg326157%28v=crm.6%29.aspx.

[65] Microsoft. IOrganizationService.Update Method (Microsoft Dynamics CRM 2013). [viitattu 17.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg326166%28v=crm.6%29.

[66] Microsoft. IPlugin.Execute Method (Microsoft Dynamics CRM 2013). [viitattu 05.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg326167(v=crm.6).

[67] Microsoft. IPluginExecutionContext Interface (Microsoft Dynamics CRM 2013). [viitattu 16.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/microsoft.xrm.sdk.ipluginexecutioncontext%28v=crm.6%29.aspx.

[68] Microsoft. IPluginExecutionContext Members (Microsoft Dynamics CRM 2013). [viitattu 20.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg306108(v=crm.6).

[69] Microsoft. IWorkflowContext Interface (Microsoft Dynamics CRM 2013). [viitattu 16.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/microsoft.xrm.sdk.workflow.iworkflowcontext%28v=crm.6%29.aspx.

[70] Microsoft. MSBuild Conditions. [viitattu 29.04.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/7szfhaft.aspx.

[71] Microsoft. Plug-in development (Microsoft Dynamics CRM 2013). [viitattu 02.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328490%28v=crm.6%29.aspx.

[72] Microsoft. Plug-in isolation, trusts, and statistics (Microsoft Dynamics CRM 2013). [viitattu 02.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg334752%28v=crm.6%29.aspx.

[73] Microsoft. Sample: Create a custom workflow activity (Microsoft Dynamics CRM 2013). [viitattu 16.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg334455%28v=crm.6%29.aspx.

[74] Microsoft. Software Development Kit for Microsoft Dynamics CRM 2013). [viitattu 06.05.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/hh547453%28v=crm.6%29.aspx.

[75] Microsoft. Supported extensions for Microsoft Dynamics CRM 2013. [viitattu 29.04.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328350%28v=crm.6%29.aspx.

[76] Microsoft. Supported extensions for Microsoft Dynamics CRM 2015. [viitattu 29.04.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328350%28v=crm.7%29.aspx.

[77] Microsoft. Supported extensions for Microsoft Dynamics CRM 2016. [viitattu 29.04.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328350%28v=crm.8%29.aspx.

[78] Microsoft. The extensibility model of Microsoft Dynamics CRM 2013. [viitattu 23.02.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg327974%28v=crm.6%29.aspx.

[79] Microsoft. The metadata and data models in Microsoft Dynamics CRM 2013. [viitattu 17.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg309434(v%3dcrm.6).aspx.

[80] Microsoft. Understand the data context passed to a plug-in (Microsoft Dynamics CRM 2013). [viitattu 02.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg309673%28v=crm.6%29.aspx.

[81] Microsoft. Understanding the Global Assembly Cache (GAC) and Managing, Copying DLLs from Global Assembly Cache. [viitattu 21.10.2018]. Saatavissa https://social.technet.microsoft.com/wiki/contents/articles/29637.aspx.

[82] Microsoft. Use LINQ to construct a query (Microsoft Dynamics CRM 2013). [viitattu 17.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328328%28v=crm.6%29.aspx.

[83] Microsoft. Use the early-bound entity classes for create, update, and delete (Microsoft Dynamics CRM 2013). [viitattu 17.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328499%28v=crm.6%29.aspx.

[84] Microsoft. Use the early bound entity classes in code (Microsoft Dynamics CRM 2013). [viitattu 14.05.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328210%28v=crm.6%29.aspx.

[85] Microsoft. Use the IOrganizationService web service to read and write data or metadata (Microsoft Dynamics CRM 2013). [viitattu 16.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg309449%28v=crm.6%29.aspx.

[86] Microsoft. Use the late bound entity class in code (Microsoft Dynamics CRM 2013). [viitattu 14.05.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg309272%28v=crm.6%29.aspx.

[87] Microsoft. Windows Workflow Foundation. [viitattu 05.05.2015]. Saatavissa https://msdn.microsoft.com/en-us/library/jj684582.aspx.

[88] Microsoft. Workflow process architecture (Microsoft Dynamics CRM 2013). [viitattu 14.05.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg309387%28v=crm.6%29.aspx.

[89] Microsoft. Write a plug-in (Microsoft Dynamics CRM 2013). [viitattu 02.03.2016]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328263%28v=crm.6%29.aspx.

[90] Microsoft. Using early binding and late binding in Automation (KB-artikkeli 245115). [viitattu 28.02.2016]. Saatavissa https://support.microsoft.com/en-gb/kb/245115.

[91] Microsoft. Abstract and Sealed Classes and Class Members (C# Programming Guide). [viitattu 28.10.2018]. Saatavissa https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members.

[92] Microsoft. XML Documentation Comments (C# Programming Guide). [viitattu 15.04.2018]. Saatavissa https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/xmldoc/xml-documentation-comments.

[93] Microsoft. Register and Deploy Plug-Ins (Microsoft Dynamics CRM 2013). [viitattu 06.10.2018]. Saatavissa https://docs.microsoft.com/en-us/previous-versions/dynamicscrm-2013/developers-guide/gg309620(v%3dcrm.6).

[94] Microsoft. What is Microsoft Dynamics CRM 2013. [viitattu 17.11.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/gg328223(v%3dcrm.6).aspx.

[95] Microsoft. IntelliSense in Visual Studio. [viitattu 16.09.2018]. Saatavissa https://docs.microsoft.com/en-us/visualstudio/ide/using-intellisense.

[96] Microsoft Developer Network. MSDN-ohje: How to: Set Debug and Release Configurations. [viitattu 15.04.2018]. Saatavissa https://msdn.microsoft.com/en-us/library/wx0123s5.aspx.

[97] Oracle Technology Network. Core J2EE Patterns – Data Access Object. [viitattu 02.05.2016]. Saatavissa http://www.oracle.com/technetwork/java/dataaccessobject-138824.html.

[98] Oracle Technology Network. Core J2EE Patterns – Transfer Object. [viitattu 02.05.2016]. Saatavissa http://www.oracle.com/technetwork/java/transferobject-139757.html.

[99] NLog-lokituskirjasto. [viitattu 28.02.2015]. Saatavissa http://nlog-project.org/.

[100] NSubstibute-mockupkirjasto. [viitattu 16.01.2015]. Saatavissa http://nsubstitute.github.io/.

[101] Oracle. Oracle NetSuite. [viitattu 10.11.2018]. Saatavissa http://www.netsuite.com.

[102] Harold Ossher & Peri Tarr. Multi-Dimensional Separation of Concerns in Hyperspace. Research Report RC 21452(96717). Yorktown Heights: Reseach Division, IBM, huhtikuuta 1999.

[103] D. L. Parnas. Information distribution aspects of design methodology. Technical report. Pittsburgh: Carnegie Mellon University, helmikuuta 1971. URL: http://repository.cmu.edu/compsci/1829/.

[104] D. L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. Communications of the ACM 15(1972)12, pp. 1053–1058.

[105] Microsoft Research. ILMerge. [viitattu 24.04.2016]. Saatavissa http://research.microsoft.com/en-us/people/mbarnett/ILMerge.aspx.

[106] Matti Rintala & Jyke Jokinen. Olioiden ohjelmointi C++:lla. Helsinki Talentum, 2005. 377 s.

[107] SalesForce. SalesForce. [viitattu 10.11.2018]. Saatavissa https://www.salesforce.com.

[108] Mark Seemann. Dependency Injection in .NET. Shelter Island Manning, 2012. 552 p.

[109] Navaid Shamsee, David Klebanov, Hesham Fayed, Ahmed Afrose & Ozden Karakok. CCNA Data Center DCICT 640-916: Official Cert Guide. Indiapolis Cisco Press, 2015. 1005 p.

[110] SuiteScript 2.0 API. 2018.2. Oracle 2018. 1371 p. [viitattu 12.09.2018]. Saatavissa URL: https://docs.oracle.com/cloud/latest/netsuitecs_gs/NSAPI/NSAPI.pdf.

[111] Systems and software engineering – Vocabulary. ISO/IEC/IEEE 24765:2010(E) (2010)s. 1–418.

[112] Peri Tarr, Harold Ossher, William Harrison & Stanley M. Sutton Jr. N Degrees of Separation: Multi-dimensional Separation of Concerns. Los Angeles, California, USA 1999, ACM. s. 107–119.

[113] Francois Valdy. ILRepack. [viitattu 24.04.2016]. Saatavissa https://github.com/gluck/il-repack.

[114] xUnit-testauskirjasto. [viitattu 01.01.2016]. Saatavissa http://xunit.github.io/.

[115] Liming Zhu & Ian Gorton. UML Profiles for Design Decisions and Non-Functional Requirements. Washington 2007, IEEE Computer Society. pp. 8–15.

A. ESIMERKKI STAATTISESTI TYYPITETYSTÄ DTO-LUOKASTA

Tässä liitteessä esitetään Dynamics CRM SDK:n sisältämän CrmSvcUtil-työkalun [45] luomaa, staattisesti tyypitetyn DTO-luokan ohjelmakoodia. Katso tarkempi kuvaus DTO-luokkien käytöstä Dynamics CRM -kehitystyössä kohdan 3.4 sivulta 22. Alla oleva ohjelmalista esittää katkelman staattisesti tyypitetyn DTO-luokan automaattisesti luodusta lähdekoodista.

```
1  /// <summary>
   /// Person with whom a business unit has a relationship, such as customer, supplier, and colleague.
3  /// </summary>
   [System.Runtime.Serialization.DataContractAttribute()]
5  [Microsoft.Xrm.Sdk.Client.EntityLogicalNameAttribute("contact")]
   [System.CodeDom.Compiler.GeneratedCodeAttribute("CrmSvcUtil", "6.0.0001.0061")]
7  public partial class Contact: Microsoft.Xrm.Sdk.Entity, System.ComponentModel.INotifyPropertyChanging,
       System.ComponentModel.INotifyPropertyChanged {

9    /// <summary>
     /// Default Constructor.
11   /// </summary>
     public Contact() :
13       base(EntityLogicalName) {
     }
15
     public const string EntityLogicalName = "contact";
17   public const int EntityTypeCode = 2;
     public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
19   public event System.ComponentModel.PropertyChangingEventHandler PropertyChanging;

21   private void OnPropertyChanged(string propertyName) {
       if((this.PropertyChanged != null)) {
23       this.PropertyChanged(this, new System.ComponentModel.PropertyChangedEventArgs(propertyName));
       }
25   }

27   private void OnPropertyChanging(string propertyName) {
       if((this.PropertyChanging != null)) {
29       this.PropertyChanging(this, new System.ComponentModel.PropertyChangingEventArgs(propertyName));
       }
31   }

33   /// <summary>
     /// Unique identifier of the contact.
35   /// </summary>
     [Microsoft.Xrm.Sdk.AttributeLogicalNameAttribute("contactid")]
37   public System.Nullable<System.Guid> ContactId {
       get {
39       return this.GetAttributeValue<System.Nullable<System.Guid>>("contactid");
       }
41     set {
         this.OnPropertyChanging("ContactId");
43       this.SetAttributeValue("contactid", value);
         if(value.HasValue) {
45         base.Id = value.Value;
         }
47       else {
```

```
        base.Id = System.Guid.Empty;
      }
      this.OnPropertyChanged("ContactId");
    }
  }

  [Microsoft.Xrm.Sdk.AttributeLogicalNameAttribute("contactid")]
  public override System.Guid Id {
    get {
      return base.Id;
    }
    set {
      this.ContactId = value;
    }
  }

  /// <summary>
  /// Type the contact's first name to make sure the contact is addressed correctly in sales calls, email, and
      marketing campaigns.
  /// </summary>
  [Microsoft.Xrm.Sdk.AttributeLogicalNameAttribute("firstname")]
  public string FirstName {
    get {
      return this.GetAttributeValue<string>("firstname");
    }
    set {
      this.OnPropertyChanging("FirstName");
      this.SetAttributeValue("firstname", value);
      this.OnPropertyChanged("FirstName");
    }
  }

  /// <summary>
  /// Type the contact's last name to make sure the contact is addressed correctly in sales calls, email, and
      marketing campaigns.
  /// </summary>
  [Microsoft.Xrm.Sdk.AttributeLogicalNameAttribute("lastname")]
  public string LastName {
    get {
      return this.GetAttributeValue<string>("lastname");
    }
    set {
      this.OnPropertyChanging("LastName");
      this.SetAttributeValue("lastname", value);
      this.OnPropertyChanged("LastName");
    }
  }

  /// <summary>
  /// Select the parent account or parent contact for the contact to provide a quick link to additional
      details, such as financial information, activities, and opportunities.
  /// </summary>
  [Microsoft.Xrm.Sdk.AttributeLogicalNameAttribute("parentcustomerid")]
  public Microsoft.Xrm.Sdk.EntityReference ParentCustomerId {
    get {
      return this.GetAttributeValue<Microsoft.Xrm.Sdk.EntityReference>("parentcustomerid");
    }
    set {
      this.OnPropertyChanging("ParentCustomerId");
      this.SetAttributeValue("parentcustomerid", value);
      this.OnPropertyChanged("ParentCustomerId");
    }
  }

  /// <summary>
  /// Shows whether the contact participates in workflow rules.
  /// </summary>
  [Microsoft.Xrm.Sdk.AttributeLogicalNameAttribute("participatesinworkflow")]
  public System.Nullable<bool> ParticipatesInWorkflow {
    get {
      return this.GetAttributeValue<System.Nullable<bool>>("participatesinworkflow");
    }
    set {
      this.OnPropertyChanging("ParticipatesInWorkflow");
      this.SetAttributeValue("participatesinworkflow", value);
      this.OnPropertyChanged("ParticipatesInWorkflow");
    }
  }

  /// <summary>
```

```
125   /// Select the payment terms to indicate when the customer needs to pay the total amount.
      /// </summary>
127   [Microsoft.Xrm.Sdk.AttributeLogicalNameAttribute("paymenttermscode")]
      public Microsoft.Xrm.Sdk.OptionSetValue PaymentTermsCode {
129     get {
          return this.GetAttributeValue<Microsoft.Xrm.Sdk.OptionSetValue>("paymenttermscode");
131     }
        set {
133       this.OnPropertyChanging("PaymentTermsCode");
          this.SetAttributeValue("paymenttermscode", value);
135       this.OnPropertyChanged("PaymentTermsCode");
        }
137   }

139   /// <summary>
      /// Shows whether the contact is active or inactive. Inactive contacts are read-only and can't be edited
           unless they are reactivated.
141   /// </summary>
      [Microsoft.Xrm.Sdk.AttributeLogicalNameAttribute("statecode")]
143   public System.Nullable<Example.DAO.ContactState> StateCode {
        get {
145       Microsoft.Xrm.Sdk.OptionSetValue optionSet =
              this.GetAttributeValue<Microsoft.Xrm.Sdk.OptionSetValue>("statecode");
          if((optionSet != null)) {
147         return ((Example.DAO.ContactState)(System.Enum.ToObject(typeof(Example.DAO.ContactState),
                optionSet.Value)));
          }
149       else {
            return null;
151       }
        }
153   }

155   /// <summary>
      /// 1:N quote_customer_contacts
157   /// </summary>
      [Microsoft.Xrm.Sdk.RelationshipSchemaNameAttribute("quote_customer_contacts")]
159   public System.Collections.Generic.IEnumerable<Example.DAO.Quote> quote_customer_contacts {
        get {
161       return this.GetRelatedEntities<Example.DAO.Quote>("quote_customer_contacts", null);
        }
163     set {
          this.OnPropertyChanging("quote_customer_contacts");
165       this.SetRelatedEntities<Example.DAO.Quote>("quote_customer_contacts", null, value);
          this.OnPropertyChanged("quote_customer_contacts");
167     }
      }
169
      /// <summary>
171   /// N:N contactorders_association
      /// </summary>
173   [Microsoft.Xrm.Sdk.RelationshipSchemaNameAttribute("contactorders_association")]
      public System.Collections.Generic.IEnumerable<Example.DAO.SalesOrder> contactorders_association {
175     get {
          return this.GetRelatedEntities<Example.DAO.SalesOrder>("contactorders_association", null);
177     }
        set {
179       this.OnPropertyChanging("contactorders_association");
          this.SetRelatedEntities<Example.DAO.SalesOrder>("contactorders_association", null, value);
181       this.OnPropertyChanged("contactorders_association");
        }
183   }

185   /// <summary>
      /// N:1 contact_customer_accounts
187   /// </summary>
      [Microsoft.Xrm.Sdk.AttributeLogicalNameAttribute("parentcustomerid")]
189   [Microsoft.Xrm.Sdk.RelationshipSchemaNameAttribute("contact_customer_accounts")]
      public Example.DAO.Account contact_customer_accounts {
191     get {
          return this.GetRelatedEntity<Example.DAO.Account>("contact_customer_accounts", null);
193     }
        set {
195       this.OnPropertyChanging("contact_customer_accounts");
          this.SetRelatedEntity<Example.DAO.Account>("contact_customer_accounts", null, value);
197       this.OnPropertyChanged("contact_customer_accounts");
        }
199   }

201 }
```

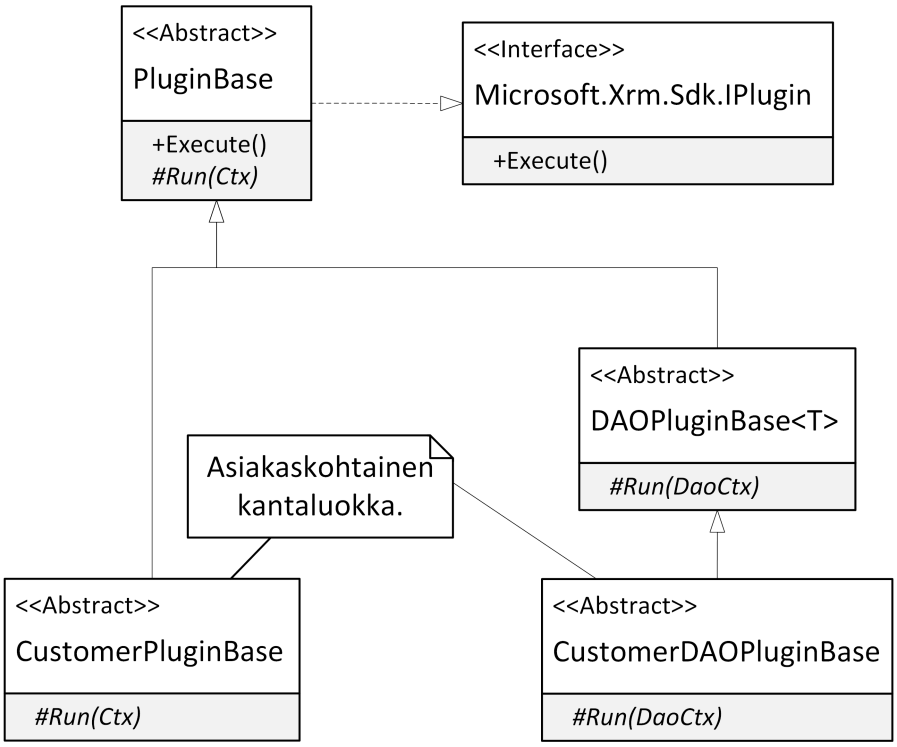
B. EXTENSIONBASE-KIRJASTON RAKENNE

Kohdassa 5.1 on kuvattu komponenttikehyksen kirjastorakenne yleisellä tasolla. Tässä liitteessä kuvataan tarkemmin komponenttikehyksen *ExtensionBase*-kirjaston sisältö. Tässä kirjastossa on liitännäisten ja räätälöityjen työnkulun vaiheiden toteutuksiin tarkoitetut abstraktit kantaluokat ja näiden sisäisesti käyttämät toteutukset. Kantaluokista tarjotaan sekä dynaamisen että staattisen ohjelmointitavan[1] mahdollistavat versiot. Liitännäisten ja räätälöityjen työnkulun vaiheiden luokkien rakenteiden välillä on joitakin tässä kohdassa käsiteltäviä eroja. Komponenttikehys tarjoaa seuraavat kantaluokat:

- *PluginBase* – liitännäisille
  - Toteuttaa Dynamics CRM:n liitännäisiltä vaaditun rajapinnan *IPlugin*.
  - Mahdollistaa vain dynaamisesti sidotun ohjelmointitavan käytön.
- *DAOPluginBase<T>* – liitännäisille
  - Toteuttaa Dynamics CRM:n liitännäisiltä vaaditun rajapinnan *IPlugin*.
  - Mahdollistaa sekä staattisesti että dynaamisesti sidotun ohjelmointitavan käytön.
  - Vaatii tyyppiparametrina T staattisesti sidotun DAO-kontekstiluokan.
- *WorkflowTemplateBase* – räätälöidyille työnkulun vaiheille.
  - Mahdollistaa vain dynaamisesti sidotun ohjelmointitavan käytön.
- *DAOWorkflowTemplateBase<T>* – räätälöidyille työnkulun vaiheille.
  - Mahdollistaa sekä staattisesti että dynaamisesti sidotun ohjelmointitavan käytön.
  - Vaatii tyyppiparametrina T staattisesti sidotun DAO-kontekstiluokan.

[1] Katso dynaaminen ja staattinen ohjelmointitapa kohdan 3.4 sivuilta 22 ja 24.

Liitännäisten osalta asiakaskohtainen kantaluokka saadaan aikaan suoraviivaisesti periyttämällä se komponenttikehyksen sisältämästä kantaluokasta *PluginBase* tai *DAOPluginBase<T>*. Liitännäisten kantaluokkien hierarkiaa selventää kuva B.1.



Kuva B.1 *Liitännäisten kantaluokkien keskinäiset suhteet UML:n luokkakaaviona.*

Mikäli käytetään staattisesti sidottua kantaluokkaa *DAOPluginBase<T>*, on asiakaskohtaiselle liitännäisen kantaluokalle erikseen määriteltävä, että se on rajapinnan *Microsoft.Xrm.Sdk.IPlugin* mukainen. Kun kantaluokassa on käytetty luokkamallia, Dynamics CRM ei jostain syystä tulkitse periytetyn luokan toteuttavan *IPlugin*-rajapintaa. .NET-kirjaston muilla työkaluilla tutkittaessa luokan havaitaan toteuttavan kyseinen rajapinta. Syy tähän tarpeeseen on epäselvä ja johtuu mahdollisesti Dynamics CRM -alustassa olevasta virheestä.

Ohjelmalistaus B.1 näyttää, miten kuvan B.1 mukainen asiakaskohtainen kantaluokka *CustomerDAOPluginBase* periytetään komponenttikehyksen tarjoamasta kantaluokasta. Ohjelmalistauksen esimerkissä käytetään sellaista komponenttikehyksen kantaluokan rakentajaa, joka mahdollistaa lokikirjoituksen, staattisesti sidotun ohjelmointitavan käytön ja erikoistetun virhekäsittelyn. Kantaluokan rakentajalle annetaan ensimmäisenä parametrina halutun lokikonfiguroinnin kapseloiva instanssi luokasta *ExtensionLoggerConfiguration*. Toisena parametrina annetaan viittaus staattisesti sidottuja DAO-kontekstiluokan[2] instansseja luovaan tehdasmetodiin. Viimeisellä parametrilla otetaan käyttöön komponenttikehyksen tarjoaman vir-

[2]Katso DAO-kontekstiluokka kohdan 3.4 sivulta 23.

hekäsittelyn oletustoteutus instantioimalla luokka *ExtensionBaseDefaultException-Handler*.

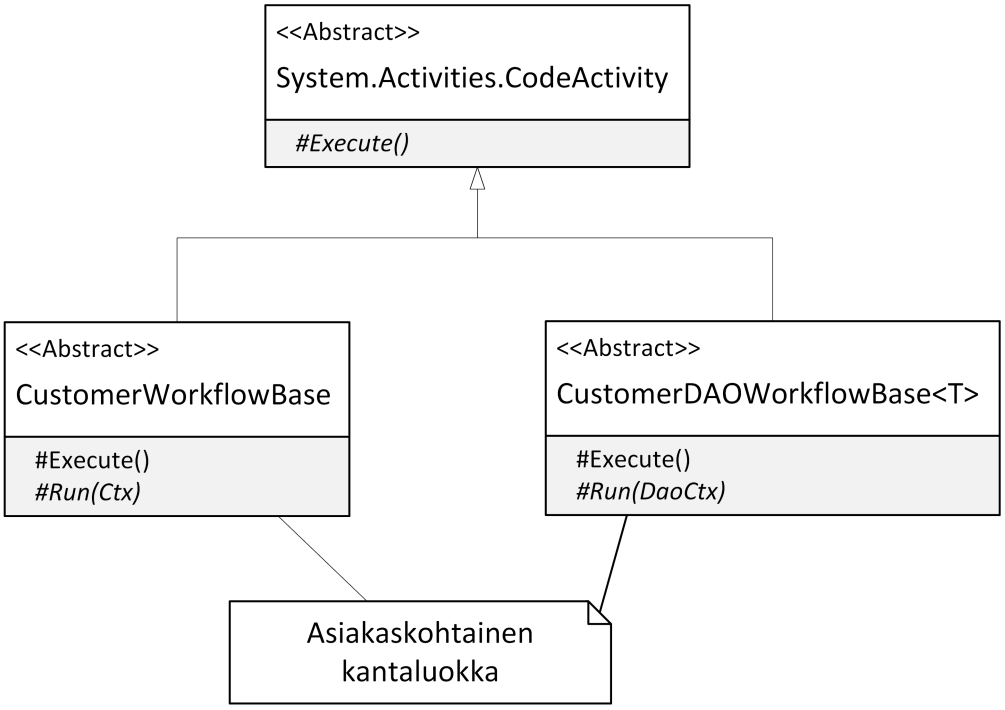
```
1  public abstract class CustomerDAOPluginBase:
     DAOPluginBase<DAOContext>, IPlugin {
3
     public CustomerDAOPluginBase(string loggerIdentifier):
5      base(new ExtensionLoggerConfiguration(LoggerFactory,
                 InitializeLogger, loggerIdentifier),
7           DaoFactoryMethod,
            new ExtensionBaseDefaultExceptionHandler()) {
9    }

11   // Alla olevat metodit on kuvattu erikoistamisen
     // havainnollistamiseksi ilman parametreja ja toteutusta.
13   private static ILogger LoggerFactory(...) {...};
     private static void InitializeLogger(...) {...};
15   private static DAOContext DaoFactoryMethod(...) {...};
   }
```

Ohjelma B.1 *Esimerkki asiakaskohtaisesti erikoistetun kantaluokan periyttämisestä komponenttikehyksen tarjoamasta kantaluokasta. Tässä havainnollistuu riippuvuuksien injektointi erikoistamisrajapinnan kautta.*

Räätälöityjen työnkulun vaiheiden osalta asiakaskohtaista kantaluokkaa ei saada konfiguroitua käyttöön niin suoraviivaisesti kuin liitännäisten tapauksessa. Monimutkaisempi konfigurointi johtuu Dynamics CRM:n vaatimuksesta periyttää räätälöidyt työnkulun vaiheet luokasta *System.Activities.CodeActivity*. Koska .NET-kielet eivät tue moniperintää [3], ei voida käyttää liitännäisten yhteydessä käytettyä suoraa periyttämistä komponenttikehyksen tarjoamasta kantaluokasta.

Suoran periytymisen asemasta räätälöityjen työnkulun vaiheille tehtiin komponenttikehykseen asiakaskohtaisen logiikan suorituksen käsittelijät. Nämä käsittelijät toteuttavat komponenttikehyksen ydintoimintaa. Käytännössä siis käsittelijät takaavat, että kohdassa 5.3 kuvattu asiakaslogiikan toteutuksen kutsuminen toimii identtisesti liitännäisten kanssa. Ratkaisua selventää kuva B.2.

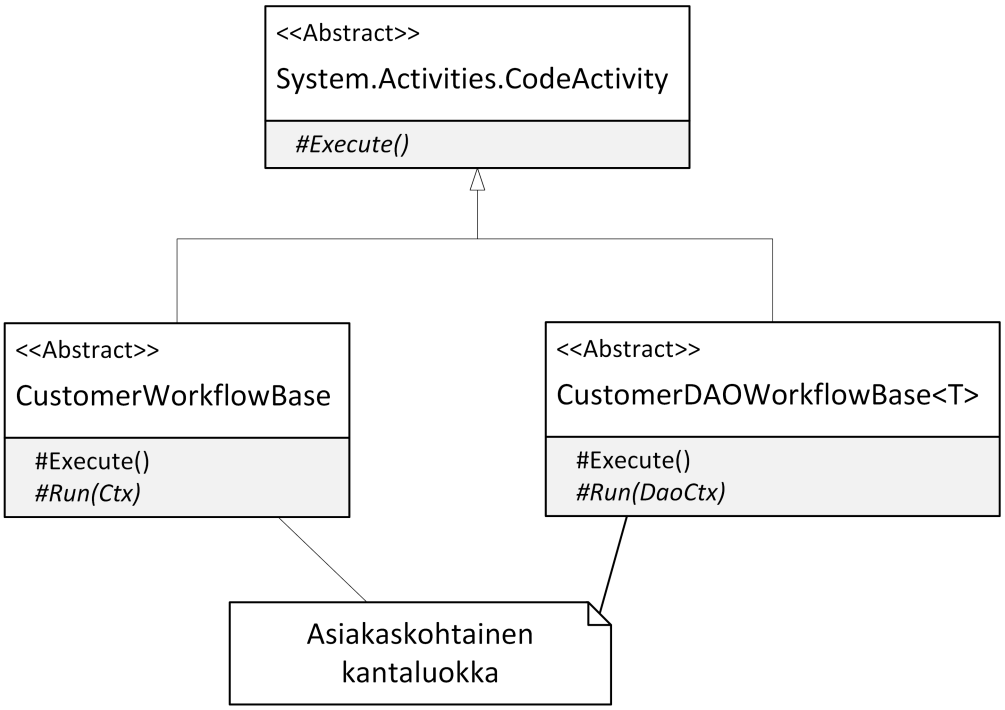
Komponenttikehys tarjoaa kantaluokat työnkulun käsittelijöille. Asiakasprojektiin pitää toteuttaa kantaluokka räätälöidyn työnkulun vaiheelle. Tämä kantaluokka periytetään luokasta *CodeActivity*. Toteutetun kantaluokan vastuulla on siirtää logiikan suoritus työnkulun käsittelijälle. Työnkulun käsittelijä periytetään komponent-



Kuva B.2 *Dynaamisesti sidotun räätälöidyn työnkulun vaiheen asiakaskohtaisen kantaluokan, komponenttikehyksen tarjoaman työnkulun käsittelijän sekä asiakaskohtaisen käsittelijän välinen suhde UML:n luokkakaaviona. Luokka CustomerWorkflowBase on asiakaskohtainen kantaluokka ja CustomerWorkflowHandler on asiakaskohtainen käsittelijäluokka.*

tikehyksen tarjoamasta kantaluokasta *WorkflowTemplateBase* tai *DAOWorkflowTemplateBase<T>*. Asiakasprojektin työnkulun kantaluokan tulee vaatia sen käyttäjää toteuttamaan metodi *Run()* kuten liitännäistenkin tapauksessa. Yhteenvetona tapahtuu siis seuraavaa: asiakaskohtainen kantaluokka siirtää asiakaslogiikan suorituksen asiakaskohtaiselle räätälöityjen työnkulkujen käsittelijälle, joka puolestaan periytyy komponenttikehyksen kantaluokasta.

Edellä kuvattu räätälöityjen työnkulun vaiheiden kantaluokkien hieman monimutkainen rakenne on tehty komponenttikehyksen käyttäjän takia. Rakenteella on haettu yhtenäistä käyttökokemusta riippumatta siitä, tehdäänkö liitännäistä vai räätälöityä työnkulun vaihetta. Komponenttikehystä käyttävän ohjelmistokehittäjän näkökulmasta asiakaskohtaisen räätälöidyn työnkulun vaiheen toteutus on kuvan B.3 mukainen. Kun tätä kuvaa verrataan liitännäisten kantaluokkiin (kuva B.1) sekä komponenttikehyksen korkeamman tason kuvaukseen (kuva 5.5), huomataan liitännäisten kantaluokkien ja räätälöityjen työnkulun vaiheiden kantaluokkien olevan käyttäjän näkökulmasta hyvin samanlaisia.



Kuva B.3 *Räätälöidyn työnkulun vaiheen kantaluokat ohjelmistokehittäjän näkökulmasta UML:n luokkakaaviona. Kuvassa B.2 näkyvät luokat CustomerWorkflowHandler ja WorkflowTemplateBase ovat sellaisia, jotka piiloutuvat ohjelmistokehittäjältä, kun hän käyttää asiakaskohtaista räätälöidyn työnkulun kantaluokkaa.*

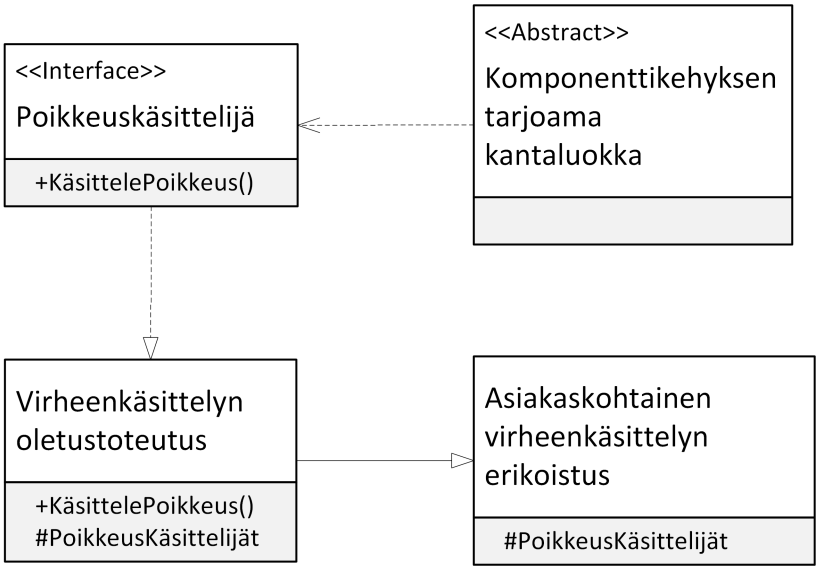
C. ERIKOISTAMISEN KÄYTTÖTAPAUSTEN OLETUSTOTEUTUKSET

Tässä liitteessä kuvataan komponenttikehyksen asiakaskohtaisen konfiguroinnin erikoistamisen yksityiskohtaisempi toteutus. Asiakaskohtaisen konfiguroinnin erikoistus on kuvattu ylätasolla kohdassa 5.2. Tässä liitteessä kerrotaan komponenttikehyksen tarjoamat erikoistamista tukevat oletustoteutukset ja miten niitä on tarkoitus käyttää komponenttikehyksen konfiguroinnin erikoistamisen käyttötapauksissa.

Lokituksen erikoistaminen tehdään toteuttamalla lokitus halutun lokituskirjaston kautta. Komponenttikehyksen rakentajalle tarjotaan tehdasmetodi uuden lokitusinstanssin saamiseksi. Lokituksessa voi tukeutua tarjottuun oletustoteutukseen, joka kirjoittaa lokin Dynamics CRM:n *ITracingService*-rajapinnan kautta. Paikallisesti asennetuissa Dynamics CRM -ympäristöissä voi käyttää esimerkiksi kirjastossa *Logger.NLog* olevaa toteutusta.

Kirjastossa *Common* on oletustoteutus ajonaikaisten virheiden käsittelyyn. Tämä toteutus ainoastaan muotoilee ilmoituksen odottamattomasta virheestä hieman Dynamics CRM -alustan oletustoteutusta paremmin. Tästä syystä toteutusta ei voi sellaisenaan suositella käytettäväksi asiakasprojekteissa. Tässä valmiissa oletustoteutuksessa on kuitenkin oma erikoistamisrajapintansa. Tämän rajapinnan avulla helpotetaan asiakasprojektikohtaisesti räätälöidyn virheenkäsittelyn käyttöönottoa. Asiakasprojekteissa voidaan erikoistaa vain halutut virheenkäsittelyn osat ja tukeutua muilta osin oletustoteutukseen. Erikoistaminen tehdään rekisteröimällä poikkeusluokkakohtaiset käsittelymetodit takaisinkutsujen avulla. Tämä tekniikka valittiin kahdesta syystä Se auttaa pitämään komponenttikehyksen virheenkäsittelyn logiikka muuttumattomampana sen erikoistamisen yhteydessä. Lisäksi valittu tekniikka poistaa tarpeen toistaa saman logiikan toteutus aina uudestaan jokaiseen asiakasprojektiin. Oletusvirheenkäsittelyn erikoistamisrajapintaa havainnollistaa kuva C.1. Ajonaikaiset virheet käsittelevän toteutuksen instanssi annetaan parametrina komponenttikehyksen rakentajalle.

Kirjastossa *Common* oleva asetustietueiden käsittelyn oletustoteutus vaatii asiakasprojektikohtaisesti tehtävän dynaamisen sidonnan konfiguroinnin. Tämä tarkoittaa



Kuva C.1 *Komponenttikehyksen tarjoamat kantaluokat vaativat virheenkäsittelyn erikoistamisen toteutuksen. Komponenttikehys tarjoaa virheenkäsittelyn oletustoteutuksen. Tämä oletustoteutus voidaan erikoistaa asiakaskohtaisesti tämän UML-luokkakaavion mukaisella rakenteella.*

asiakasprojektissa käytettävän asetusentiteetin nimen sekä tuon entiteetin avain- ja arvo-kenttien nimien välittämistä parametrina asetustietueiden käsittelyn oletustoteukselle. Komponenttikehykselle annetaan parametrina viittaus tehdasmetodiin, jolla saa pyydettyä asetustietueita käsittelevän toteutuksen. Komponenttikehys käyttää annettuja entiteetin ja kenttien nimiä käsitelleessän Dynamics CRM -järjestelmässä olevia asetustietueita.

Sekä liitännäisten että räätälöityjen työnkulun vaiheiden kantaluokista tarjotaan dynaamisen ja staattisen ohjelmointitavan[1] käytön mahdollistavat versiot. Näiden kantaluokkien eroja on käsitelty liitteessä B. Staattinen ohjelmointitapa otetaan käyttöön ja erikoistetaan kantaluokkien luokkamalleilla. Tässä on käytetty C#-kielen *generics* -ominaisuutta. Tyyppiparametrilla yksilöidään komponenttikehyksen käyttäjälle tarjottava DAO-kontekstiluokka[2]. Erikoistamisen näkökulmasta dynaamisen ja staattisen ohjelmointimallin kantaluokkien ainoa ero liittyy käytettävän DAO-kontekstiluokan kuvaamiseen. Staattinen kantaluokan versio vaatii tyyppiparametrina DAO-kontekstiluokan. Lisäksi vaaditaan rakentajan parametrina viittaus tehtaaseen, jonka vastuulla on DAO-kontekstiluokan ilmentymien luonti. Näiden kahden tiedon avulla komponenttikehyksen kantaluokka mahdollistaa staattisesti sidotun ohjelmointitavan käytön asiakasprojektin konkreettisissa toteutuksissa.

[1]Katso dynaaminen ja staattinen ohjelmointitapa kohdan 3.4 sivuilta 22 ja 24
[2]Katso DAO-kontekstiluokka kohdan 3.4 sivulta 23.

D. KOMPONENTTIKEHYSKONTEKSTIEN TEKNINEN KUVAUS

Komponenttikehyskonteksti on esitelty yleisellä tasolla kohdassa 5.4. Tässä liitteessä kuvataan komponenttikehyskontekstin eri variaatiot ja niiden erot yleiskuvausta teknisemmin. Lisäksi kuvataan variaatioiden keskinäiset suhteet.

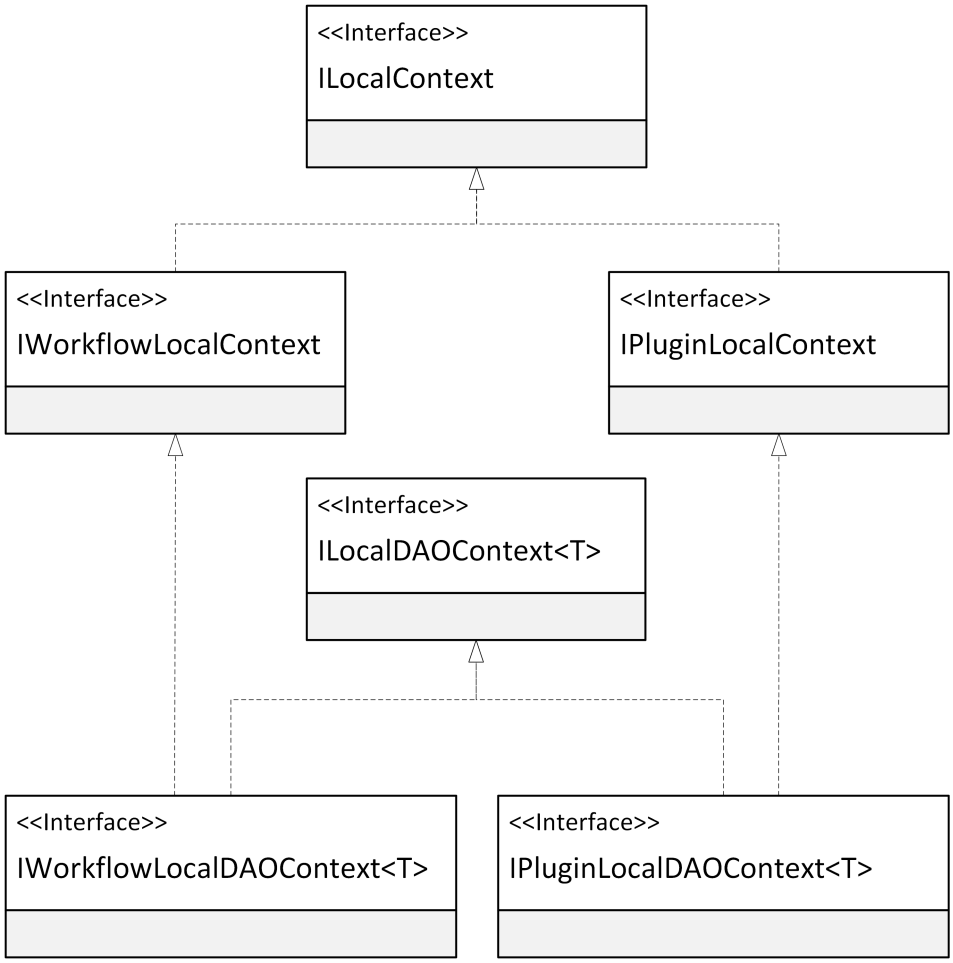
Komponenttikehyskonteksti on luokasta *ILocalContext* periytetty rajapintainstanssi. Tämä instanssi toimii sekä fasadina että siltana. Rajapinta *ILocalContext* tarjoaa fasadina useita tietoja ohjelmistokehittäjän käytettäväksi ja hyödynnettäväksi. Rajapinnasta on periytetty omat variaatiot liitännäisten ja räätälöityjen työnkulun vaiheiden *Run()*-metodeille välitettäväksi. Tässä ominaisuudessa *ILocalContext* toimii siltana piilottaen konkreettisten toteutustekniikoiden erot. Metodille välitettävä *ILocalContext*-instanssi on kohdassa 5.4 kuvattu komponenttikehyskonteksti.

Kaikki komponenttikehyskontekstin variaatiot periytetään rajapinnasta *ILocalContext*. Liitännäisille tarkoitettu rajapinnan variaatio on *IPluginLocalContext*. Räätälöityjen työnkulun vaiheiden vastaava rajapinta on *IWorkflowLocalContext*. Nämä komponenttikehyskontekstit ovat hyvin samanlaiset. Kummankin kautta tarjotaan mahdollisuus käyttää Dynamics CRM -alustan suorituskontekstiin liittyviä vakiorajapintoja *IExecutionContext*, *IOrganizationService* ja *ITracingService*. Näiden komponenttikehyskontekstirajapintojen yhteiset osat on määritelty niiden perimässä *ILocalContext*-rajapinnassa. Komponenttikehyskontekstien yhteistä toiminnallisuutta on myös jokaisen kontekstin kautta kirjoitettavan lokiviestin liittäminen automaattisesti *ITracingService*-rajapinnan tarjoamaan viestivirtaan.

Komponenttikehyskontekstin staattisen ohjelmointitavan[1] variantit ovat *IPluginLocalDAOContext<T>* ja *IWorkflowLocalDAOContext<T>*. Näistä ensimmäinen on periytetty liitännäisille tarkoitetusta dynaamisen ohjelmointitavan komponenttikehyskontekstista *IPluginLocalContext*. Jälkimmäinen puolestaan on tarkoitettu räätälöidyille työnkulun vaiheille ja se periytyy rajapinnasta *IWorkflowLocalContext*.

[1] Katso staattinen ohjelmointitapa kohdan 3.4 sivulta 24.

Komponenttikehyskontekstivariaatioita ja niiden keskinäisiä suhteita selventää kuva D.1.



Kuva D.1 *Komponenttikehyksen liitännäisille ja räätälöidyille työnkulun vaiheille tarjoamien komponenttikehyskontekstivariaatioiden keskinäiset suhteet UML:n luokkakaaviona. Rajapinnat ILocalContext ja ILocalDAOContext<T> kuvaavat periytettyjen rajapintojen yhteisesti jakaman toiminnallisuuden.*

Liitännäisten komponenttikehyskonteksteissa Dynamics CRM -alustan suorituskonteksti tarjotaan rajapinnan *Microsoft.Xrm.Sdk.IPluginExecutionContext* mukaisena yleistetymmän rajapinnan *Microsoft.Xrm.Sdk.IExecutionContext* asemasta. Tämän rajapinnan tietojen tarjoaminen on tärkeää, jotta komponenttikehyksen käyttäjä saa halutessaan käyttöön kaiken Dynamics CRM -alustan tarjoaman suorituskontekstitiedon.

Liitännäisia vastaavasti räätälöityjen työnkulun vaiheiden Dynamics CRM -alustan suorituskonteksti tarjotaan rajapinnan *Microsoft.Xrm.Sdk.IWorkflowContext* kautta yleistetymmän rajapinnan *Microsoft.Xrm.Sdk.IExecutionContext* asemasta. Komponenttikehyskontekstin kautta tarjotaan myös Dynamics CRM:n työnkulkujen suoritusalustana käyttämän Windows Workflow Foundationin käyttämä suorituskon-

tekstiluokka. Konseptimielessä tämä suorituskonteksti on melko samanlainen asia kuin liitännäisten yhteydessä mainittu suorituskonteksti. Työnkulkujen suoritustusalustan käyttämä suorituskontekstiluokka on tyypiltään *System.Activities.CodeActivityContext*. Tätä luokkaa käytetään välittämään tietoja räätälöidyn työnkulun vaiheen ja sitä kutsuvan työnkulkumoottorin kesken.